TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

VEERA SOMERO

WEB-SOVELLUKSEN REGRESSIOTESTAUKSEN AUTOMATISOINTI

Diplomityö

Tarkastaja:
professori Hannu Jaakkola
Tarkastaja ja aihe hyväksytty Talouden ja rakentamisen tiedekuntaneuvoston kokouksessa 4. toukokuuta 2016

TIIVISTELMÄ

VEERA SOMERO: Web-sovelluksen regressiotestauksen automatisointi
Tampereen teknillinen yliopisto
Diplomityö, 44 sivua, 0 liitesivua
Kesäkuu 2016
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotekniikka
Tarkastaja: professori Hannu Jaakkola

Avainsanat: testaus, testiautomaatio, regressiotestaus, robot framework

Testauksen ja testiautomaation merkitys on kasvanut ohjelmistoprojekteissa vuosikymmenen aikana, koska sovelluksista on tullut monimutkaisempia ja niiden käyttäjämäärät ovat kasvaneet.

Tässä diplomityössä automatisoitiin web-sovelluksen regressiotestaus, minkä tarkoituksena on nopeuttaa ja parantaa laadunvarmistusta. Työ jakaantuu kahteen osaan: teoriaan ja käytäntöön.

Aluksi työssä esitellään testauksen teoriaa lähteiden kautta, jonka jälkeen perehdytään Robot Framework automaatiotestaustyökaluun. Lopuksi esitellään ja analysoidaan työn tulokset eli syntyneet automaatiotestit.

Työssä automatisoidut testitapaukset olivat: uuden asiakastilin rekisteröinti, kirjautuminen asiakas-sivulle, ostoksen tilaaminen ja navigointi sovelluksen jokaiselle sivuille. Työn lopputuloksena syntyi neljä automatisoitua testiä.

ABSTRACT

VEERA SOMERO: Automated regression testing of a web application
Tampere University of Technology
Master of Science Thesis, 44 pages, 0 Appendix pages
June 2016
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Professor Hannu Jaakkola

Keywords: testing, test automation, regression testing, robot framework

The necessity of testing and test automation has grown in software projects in a decade. This is because softwares have evolved more complex and number of users have grown.

In this thesis was automated regression testing of a web application. The Purpose of regression testing is to speed up and improve the quality assurance.
Work divides in two parts: Theory and practice.

In the beginning of this work is introduced some theory of testing throught sources. After that work is oriented into Robot Framework automation testing tool. In the end is introduced and analyzed works results in test automation data.

Automated test cases in this work were: Registering as a new customer, logging in to customer's profile, ordering a product and navigating throught every page of an application. Results of the thesis was four automated tests.

ALKUSANAT

Tämän diplomityön aihe lähti omasta halusta oppia uusia asioita ohjelmiston testauksesta ja sen automatisoinnista. Näin ohjelmistokehittäjän näkökulmasta testaus on hyvin tärkeä osa-alue ohjelmistokehityksessä. Tämä työ antaa uuden näkökulman omalle ohjelmistokehitykselle, kuinka ohjelmistoa voisi kehittää enemmän testaajan perspektiivistä.

Tietotekniikan maisterin opinnot ovat olleet minulle yksi haaveistani jo lukioajalta ja olen etuoikeutettu, kun olen saanut toteuttaa haaveeni.

Työni ohjaajaa Professori Hannu Jaakkolaa haluan kiittää asiantuntevasta ja edesauttavasta ohjauksesta.

Lopuksi haluan kiittää aviomiestäni, perhettäni ja läheisiä ystäviäni siitä, että ovat olleet tukenani koko opintojeni ajan.

Porissa, 15.4.2016

Veera Somero

SISÄLLYSLUETTELO

KUVALUETTELO

LYHENTEET JA MERKINNÄT

HTML	engl. Hypertext Markup Language, hypertekstin merkintäkieli
SUT	engl. System under test, testattava ohjelma
TDD	engl. Test-driven development, testivetoinen kehitys
URL	engl. Uniform Recourse Identifier, merkkijono, jota käytetään osoittamaan WWW-sivuja
XML	engl. Extensible Markup Language, tietynlaisen merkintäkielen yläkäsite tai standardi

1. JOHDANTO

Testaaminen on ohjelmistokehityksessä tärkeä osa-alue. Nykyään käyttäjien määrä sovelluksissa on niin suuri, että kriittisyys on kasvanut sekä sovelluksista on tullut paljon monimutkaisempia. Nämä johtavat siihen, että sovelluksen laadunvarmistamisen resursseja on nostettava ja testauksen merkitys ohjelmistokehityksessä on kasvanut.

Testausta tapahtuu niin kauan kuin ohjelmistokehitystä ja nykyään ohjelmistoprojekteissa testaukseen kulutettu aika noin puolet siitä, mitä kulutetaan aikaa ohjelmistokehitykseen. Ohjelmistotestauksen resursseja pystytään käyttämään tehokkaammin dokumentoimalla testitapaukset ja käyttämällä vanhoja testitapauksia pohjana uusille testitapauksille. On hyvä muistaa tosiasia, että mitä myöhemmässä vaiheessa virheet havaitaan, sitä kalliimmaksi niiden korjaaminen tulee.

Web-sovellukset ovat muuttunut vuosikymmenien aikana staattisesta HTML-sivusta työpöytäsovelluksiin. Ennen internetistä etsittiin tietoa, kun nyt tiedon etsimisen lisäksi käyttäjä suorittaa toimintoja sovelluksien kautta. Tämä muutos on helpottanut käyttäjiä, kun ei enää tarvitse miettiä, mitä alustaa käyttää. Ainoa vaatimus on käyttää nykyaikaista selainta. Sovelluskehittäjien puolelta tämä on vaatinut suurempia ponnistuksia, koska avoin kaikilla käytettävissä oleva sovellus tuo uusia tietoturva- ja suorituskykyongelmia.

Testauksen automatisointi on myös yksi tapa tehostaa testausta ja vähentää manuaalista työtä. Vaikka testauksen automatisointi ei ole helppoa, se maksaa pidemmässä ajassa itsensä takaisin ja vähentää manuaalisessa testauksessa syntyneitä virheitä. Hyviä automatisoitavia testejä ovat kaikki usein toistettavat testit, kuten regressiotestaus, joka on hyvä suorittaa aina ennen uuden version päivittämistä.

Tässä diplomityössä luodaan automaatiotestit web-sovelluksen regressiotesteille ja käsitellään testauksen ja automaatiotestauksen merkitystä ja tekniikoita lähteiden avulla. Työ rakentuu kahteen – teoriaan, käytäntöön. Luvut 2-4 käsittelevät teoriaa ja luvut 5-6 käytäntöä.

Luvussa kaksi käsitellään testauksen merkitystä ja esitellään testauksen eri tapoja ja menetelmiä. Kolmannessa luvussa syvennytään tarkemmin regressiotestaukseen, jossa käsitellään yleisesti regressiotestauksesta ja perustellaan miksi se on hyvä automatisoitava kohde. Neljännessä luvussa perehdytään testiautomaatioon ja esitellään eri työkaluja web-sovelluksen automaatiotestaukseen.

Viidennessä luvussa siirrytään käytäntöön ja perehdytään työssä käytettävään automaatiotestauksen teknologiaan ja esitellään testiautomaatiotyökalun asennusvaiheet. Kuudennessa luvussa esitellään työn tulokset eli testiautomaatiotapaukset ja analysoidaan onnistuneita ja epäonnistuneita tapauksia sekä esitellään esimerkki käyttö- ja testitapaus. Luvussa seitsemän esitellään työn aikana syntyneitä jatkokehitysideoita. Viimeisessä kahdeksannessa luvussa kootaan yhteenveto työstä.

2. OHJELMISTOTESTAUS

2.1 Mitä testaus on?

Testauksen perusperiaate on testata, että sovellus toimii kuin se on määritelty. Glenford J. Myers on todennut (Myers, 2012) jo vuonna 1979, että testauksessa on kyse prosessista löytää sovelluksesta virheitä. Haikalan ja Märijärven mukaan (Haikala & Märijärvi, 2006) arkikielessä testauksesta saa sellaisen kuvan, että se olisi mitä tahansa kokeilua, mutta oikeasti testauksessa on kyse systemaattisesta ja suunnitellusta prosessista.

Myers tuo (Myers, 2012) esille, että monesti testaajien tavoitteena on osoittaa, että sovellus toimii oikein. Tästä syystä vältetään helposti testitapauksia, jotka tuovat esille sovelluksen virheitä ja testauksen lisäarvo jää saavuttamatta. Testauksen onnistumiseen vaikuttavat testaajien asenne ja oikea näkökulma testauksen tavoitteisiin. Testaus on onnistunut silloin, kun ohjelmistosta on löydetty virheitä, jotka voidaan korjata. Taas epäonnistunut testaus, joka ei tuo esille virheitä, on täysin tehotonta.

Nykyään testaus määritellään laajemmin, jolloin testaus käsittää kaikki ne menetelmät, jotka parantavat ja mittaavat sovelluksen laatua (Haikala & Märijärvi, 2006), jolloin myös tarkastukset ja ohjelmakoodin staattista analysointia pidetään testauksena. Tässä työssä testaus rajataan tarkoittamaan sovellukseen tapahtuvaa testausta.

Monet ohjelmistokehittäjät ovat varmasti turhautuneet, kun asiakkaat raportoivat sovelluksesta löytyneistä virheistä. Laajojen ja huolellisten testauksien ja tuhansien erilaisten testitapausten jälkeen löytyy tuotannosta vikoja, joita ei ole testausvaiheessa havaittu. James Whittakerin (Whittaker, 2000) mielestä syitä voivat olla seuraavat: käyttäjä suorittaa testaamatonta koodia, käyttäjä käyttää sovellusta eri lailla kuin testauksessa, käyttäjä syöttää arvoja, joita ei ole testattu tai käyttäjän ympäristöä ei ole testattu.

Testaaminen voi olla kehitysprojektissa haastavin ja eniten työllistävä vaihe, mutta se kuitenkin antaa suuren tuen onnistumiselle, ja luo näin laadukkaan lopputuloksen. James Bachin mukaan (Bach, 1999) testausvaihe epäonnistuu useammin kuin kehitysvaihe, koska usein yritykset eivät käytä testaamiseen tarpeeksi resursseja.

Täydellistä testausta ei ole olemassa, koska sovelluksen virheettömyyttä ei kyetä kokonaan testaamaan. Ne sisältävät monimutkaisia syötekombinaatioita, joiden läpikäynti on mahdotonta. Kuitenkin tärkeätä on löytää virheet mahdollisimman aikaisessa vaiheessa, jolloin niistä aiheutuvat lisätyöt voidaan minimoida.

2.2 Testausprosessi

Testausprosessi etenee askel kerrallaan kuin mikä tahansa ohjelmistokehitys ja on näin oleellinen osa sitä. Huolellinen testaus vie nykyään aikaa, jolloin tarvitaan suunniteltu syystemaattinen prosessi. Ohjelmistotestaus voidaan suorittaa (Dustin, 2002) silloin, kun toiminallisuus on valmis tai testata yhdessä ohjelmiston kehityksen kanssa, jolloin testaus on jatkuva prosessi.

Testaustausprosessi koostuu syklistä (Pan, 1999) ja sen vaiheista. Testausprosessi alkaa vaatimusanalyysistä, jossa testaajat työskentelevät ohjelmistokehittäjien kanssa. Vaatimusanalyysissä selvitetään, mitkä asiat ovat testattavissa ja millä parametreilla.

Tämän jälkeen aloitetaan testauksen suunnittelu, jossa luodaan testausmalli, joka todentaa, että sovellus toimii oikein ja varmistetaan testaukseen käytettävät resurssit. Testaussuunnitelman jälkeen aloitetaan kehittämään testitapauksia, testidataa ja testimenetelmiä. Kun testit on suunniteltu, aloitetaan testien ajaminen. Testiajojen aikana testaajat raportoivat mahdollisista virheistä ohjelmistokehittäjille (Pan, 1999).

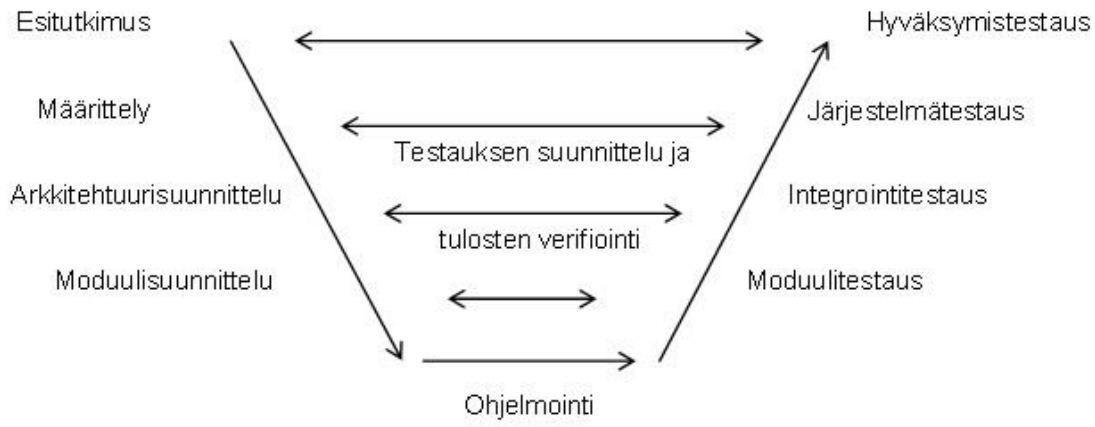
Testiajojen jälkeen luodaan testiraportit ja analysoidaan ne. Testitulokset analysoidaan ohjelmistokehittäjien ja asiakkaan kanssa, jotta voidaan pohtia, mitkä virheet korjataan, hylätään tai lykätään myöhempään ajankohtaan. Näiden vaiheiden jälkeen testeissä ilmenneet virheet uudelleen testataan korjauksien jälkeen (Pan, 1999).

Viimeinen vaihe ennen testauksen loppua on, järjestelmätestaus, jossa testataan sovellus kokonaisuudessaan määrittelydokumentissa luotujen käyttötapauksien mukaisesti. Testausprosessin viimeisin vaihe on testauksen lopettaminen, jossa arkistoidaan testauksen aikana olevat lokit, tulokset, suoritetut toiminnot ja dokumentit (Pan, 1999).

2.3 Testaustasot

Testaustasot ovat osa testauksen kokonaista prosessia ja ne kuvataan yleensä V-mallina, joka on testauksen prosessimalli. Testauksien jakaminen eri tasoihin, tekee siitä tehokkaampaa ja tarkempaa, koska tällöin testaukset voidaan kohdentaa eri tyyppisiin virheisiin ohjelmistokehityksen aikana.

Kuvassa 1 on esitetty (Korjus, 2015) testauksen V-malli, jossa eri testaustasojen suunnittelu tapahtuu samanaikaisesti ohjelmistosuunnittelun kanssa (Haikala & Märijärvi, 2006). Kuvan 1 mukaisesti moduulitestaus suunnitellaan moduulisuunnitteluvaiheessa, integraatiotestaus arkkitehtuurisuunnitteluvaiheessa ja järjestelmätestaus määrittelyvaiheessa.



Kuva 1. *Testauksen V-malli.* (Haikala & Märijärvi, 2006)

Testauksen alimmat tasot eli moduuli- ja integraatiotestaus suoritetaan ensin ja yleensä ne suorittavat ohjelmistokehittäjät, koska niiden tavoitteena on eliminoida ohjelmointi- ja suunnitteluvirheitä. Järjestelmä- ja hyväksymistestausta kutsutaan ylemmäksi tasoksi ja ne suoritetaan sovelluksen toteutusvaiheen jälkeen. Ylemmän tason testaukset suorittavat testaukseen erikoistuneet ammattilaiset.

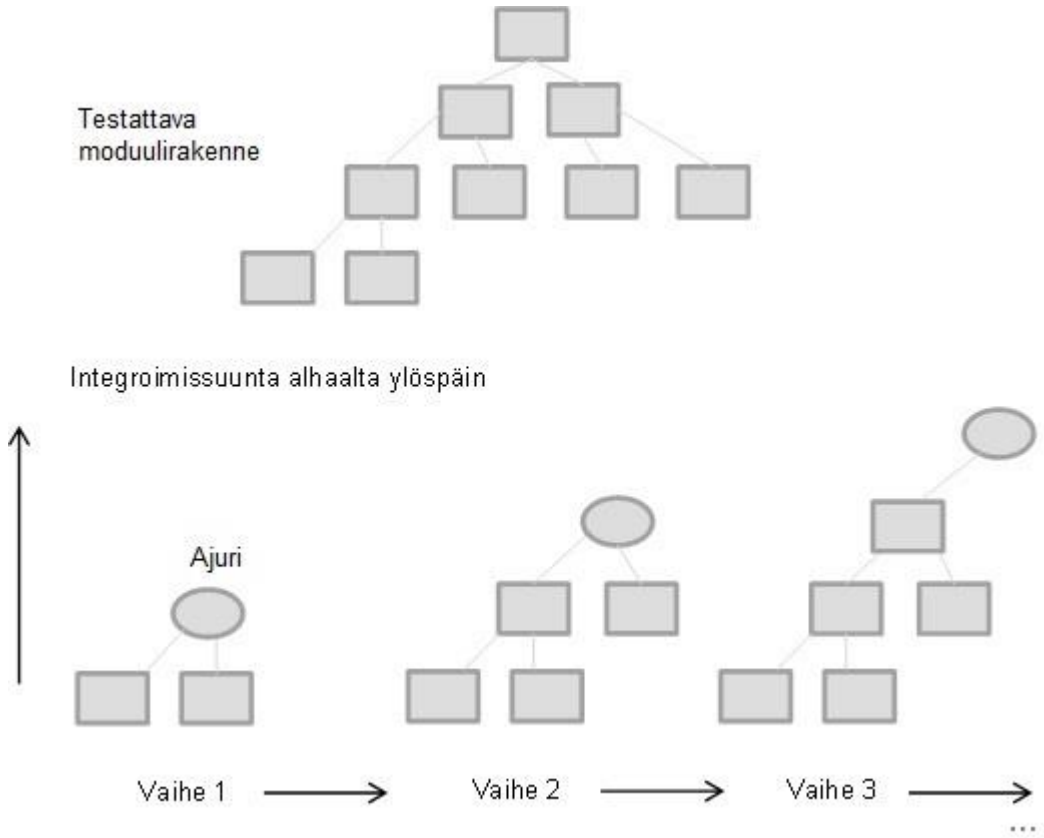
2.3.1 Moduulitestaus

Moduulitestaus on testaustason alin taso ja sitä kutsutaan myös komponenttitestaukseksi ja yksikkötestaukseksi. Moduulitestauksessa testataan 100-1000 koodirivistä koostuvaa moduulia tai komponenttia, jonka testaaja on yleensä moduulin toteuttaja. Testit tehdään heti kehitystyön jälkeen ympäristöjen yksikkötestaus-työkalujen avulla. Yksikkötestejä voidaan tehdä yhtäaikaisesti eri moduuleihin toisistaan riippumatta (Haikala & Märijärvi, 2006).

Moduulitestauksen tavoite on löytää ohjelmiston rakenteesta ja loogisesta toteutuksesta virheitä ja varmistaa, että se toimii spesifikaatioiden mukaisesti. Moduulitestauksen yksi toteutustapa on testivetoinen kehitys (TDD), jossa kirjoitetaan ensiksi testit, jotka eivät mene läpi ja tämän jälkeen luodaan toiminallisuus, jonka testit läpäisevät (Koskela, 2007). Olemassa olevat testit auttavat ylläpitovaiheessa tehtävien muutosten tekemistä. Yksi uusi muutos voi rikkoa toiminnollisuuksia ja näin ollen olemassa olevat moduulitestit ilmoittavat siitä.

2.3.2 Integraatiotestaus

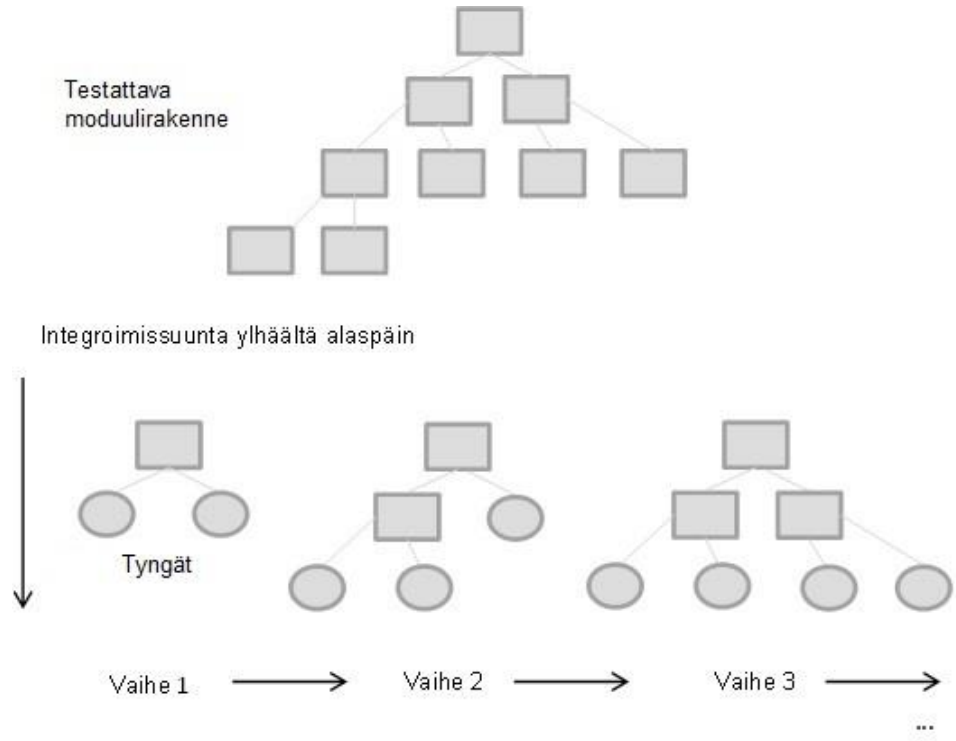
Integraatiotestauksessa on moduulitestauksesta seuraava taso, jossa tutkitaan komponenttien välisiä rajapintoja ja toimivuutta. Testit tulee suunnitella komponenttien rajapintamäärittelyiden mukaisesti, jotta virheitä voidaan löytää. Integraatiotestin etenemistapoja ovat kokoava integrointi (bottom up) tai jäsentävä integrointi (top down) (Haikala & Märijärvi, 2006).



Kuva 2. *Kokoava integrointi* (Korjus, 2015)

Kokoavan integrointi aloitetaan alimmista komponenteista, joilla ei ole riippuvuuksia ja edetään kuvan 2 mukaisesti alhaalta ylöspäin. Komponentit voidaan valita vapaasti, kunhan alemman tason komponentit on kutsuttu ennen sitä kutsuvaa ylemmän tason komponenttia (Korjus, 2015).

Jäsentävä integroinnissa etenemissuunta on kuvan 3 mukaisesti ylhäältä alaspäin, jossa testaus aloitetaan ylimmästä tai keskeisemmästä komponentista. Testattavaa komponenttia varten kehitellään testityngät, jotka korvataan myöhemmin valmiilla komponenteilla, kun sovelluksen integraatio on valmis. Jäsentävä integrointi on usein raskasta, koska testitynkiä joudutaan muokkaamaan useaan otteeseen, jotta kaikki testitapaukset saadaan testattua (Korjus, 2015).



Kuva 3. *Jäsentävä integrointi* (Korjus, 2015)

Integraatiotestauksen toteutustapa valitaan projekteissa tapauskohtaisesti. Jos virhealttiit moduulit ovat painottuneet alimmille tasoille, silloin kokoava integrointi on parempi valinta. Jäsentävän integroinnin etu on se, että lopullista sovellusta voidaan demonstroida jo aikaisessa vaiheessa, jolloin sitä voidaan käyttää hyödyksi esimerkiksi käytettävyys testeissä (Myers, 2012).

2.3.3 Järjestelmätestaus

Järjestelmätestauksessa on kyse koko sovelluksen testauksesta ja sen tuloksia verrataan ohjelmiston toiminnallisuuden määrittelyyn. Tässä tasossa olevien testien suorittajat ovat ohjelmiston kehityksen ulkopuoliset testaajat, jotka ovat testaukseen erikoistuneita ammattilaisia. Järjestelmätestauksessa testataan sovelluksen toiminallisuuden lisäksi ei-toiminallisia ominaisuuksia ja tutkitaan sen ja määrittelydokumenttien ristiriitoja. Ei-toiminallisia ominaisuuksia testaavat kuormitustestit, luotettavuustestit, asennustestit ja käytettävyystestit (Haikala & Märijärvi, 2006).

Toiminnallisuuden testauksessa verrataan olemassa olevaa sovelluksen toiminallisuuden määrittelydokumentteihin. Tarkoituksena on löytää ristiriitoja sovelluksen ja toiminallisuuden spesifikaatioiden väliltä. Spesifikaatioissa toiminnallisuus on selitetty yksityiskohtaisesti käyttäjän näkökulmasta ja niitä käyttävät ohjelmistosuunnittelijat kehitystyössään. Testauksen suorittavat kehitystyöstä ulkopuoliset testauksen ammattilaiset, jotka analysoivat ja käyttävät sovellusta kuin sen loppukäyttäjät. Toiminallisuus testaukset voidaan aloittaa heti toiminallisuuksien valmistuttua tai vasta yksikkö- ja integraatiotestauksen jälkeen (Myers, 2012).

Kuormitustestissä testataan sovelluksen käyttäytymistä eri kuormituksilla, kuten samanaikaisten käyttäjien määrä tai käsiteltävien tiedon määrää. Tässä testauksessa pyritään myös selvittämään sen maksimi taso stressi- ja rasitusteisteillä. Stressi- ja rasitusteisteissä pyritään kuormittamaan sovellus huippuun, jotta voidaan tutkia, kuinka se toimii niissä tilanteissa ja mikä on sen maksimistason kapasiteetti (Myers, 2012).

Luotettavuustestissä pyritään analysoimaan sovelluksen vikoja ja häiriöitä. Näissä pyritään keskittymään häiriöiden esiintymistiheyteen ja vakavuuteen. Testauksen lopputuloksena voidaan todentaa, että sovelluksen luotettavuus vastaa vaatimusmäärittelyssä asetettuja tavoitteita (Kit, 1995).

Asennustestauksen tarkoituksena on varmistaa asennusohjeiden – ja järjestelmien toimivuus ja testata sen asennettavuus. Tämä on tärkeä osa testausprosessia, koska se voi estää varsinaisen ja hyvin testatun ohjelmiston käytön (Myers, 2012).

Käytettävyystestauksessa pyritään testaamaan, kuinka hyvin loppukäyttäjät suoriutuvat sovelluksen käytöstä. Käytettävyystestejä suorittavat ovat yleensä yrityksen ulkopuolella olevia käyttäjiä ja heillä yleensä käytössä ohjelmiston prototyyppi. Koetilanteessa testaajien käyttäytymistä valvotaan ja analysoidaan sen jälkeen tulokset. Tulokset ovat arvokasta informaatiota kehittäjille, koska valmiin sovelluksen tulee olla mahdollisimman käyttäjäystävällinen (Haikala & Märijärvi, 2006).

Järjestelmätestauksessa havaitut virheiden korjaukset tulee aina kalliiksi, koska niiden korjaus voi aiheuttaa muutoksia useimpiin moduuleihin. Tämän lisäksi virheiden korjauksista voi koitua vielä uusia korjaustarpeita. Tällöin uusien virheiden testauksessa pitää testata kaikki moduulit ja koko sovellus uudelleen. Tällaista uudelleentestausta kutsutaan regressiotestauksessa, johon tutustutaan tarkemmin 3. luvussa (Haikala & Märijärvi, 2006).

2.3.4 Hyväksymistestaus

Hyväksymistestaus on viimeinen testaustaso, jossa testataan, onko sovellus asiakkaan vaatimusten mukainen. Se on V-mallissa ensimmäinen vaihe testien suunnittelussa ja viimeinen vaihe testien suorituksessa. Tässä testauksessa ensisijainen tavoite ei ole enää löytää virheitä, vaan demonstroida tuotantoon menevää sovellusta. Jos virheitä löytää tässä vaiheessa ohjelmistokehitystä, niiden korjaaminen voi tulla hyvin kalliiksi.

Hyväksymistestauksessa parhaimman lopputuloksen saa, jos testauksen suunnittelevat sovelluksen loppukäyttäjät, koska he ymmärtävät sen vaatimukset ja siihen liittyviä riskit. Loppukäyttäjät yleensä toimittavat realistisen ympäristön sekä suorittavat testejä määrittelydokumentissa olevien käyttötapauksien mukaisesti. Kun käyttötapaukset on kyetty käymään läpi lopullisen sovelluksen kanssa, on hyväksymistestaus suoritettu (Katara, 2011).

2.4 Testausmenetelmät

Testausmenetelmät ovat tapa testata sovellusta ja ne ohjaavat koko testausprosessia. Hyvä testausmenetelmä tukee koko ohjelmistoprojektia, koska sen tavoitteena on määrittää sekä antaa tarvittavat tiedot, joita tarvitaan testausprosessissa. Testausmenetelmät jaetaan yleisesti ottaen kahteen luokkaan, joita ovat mustalaatikko (engl. Black-box –testing) – ja lasilaatikkotestaukseen (engl. White-box –testing). (Patton, 2001).

Mustalaatikko –ja lasilaatikkotestauksen lisäksi menetelmät jaetaan staattiseen ja dynaamiseen testaukseen. Dynaamisessa testauksessa keskitytään sovelluksen testaukseen ja pyritään sovellusta ajamalla löytämään virheitä. Staattisessa testauksessa taas pyritään sovellusta ajamatta löytämään virheitä tarkkailemalla suunnitelmaa, arkkitehtuuria ja koodia (Patton, 2001).

2.4.1 Mustalaatikkotestaus

Mustalaatikkotestaus on toinen testausmenetelmistä ja sen testitapaukset valitaan sovelluksen spesifikaation mukaan ilman sovellukseen tutustumista. Tämä testaus edellyttää sovelluksen kunnollista määrittelyä ja suunnittelua. Kuvan 4 mukaisesti testauksessa sovellukselle annetaan haluttu syöte, jonka jälkeen sovellusta tutkitaan sen tuottamien tuloksien perusteella (Haikala & Märijärvi, 2006).

Kuva 4. *Mustalaatikkotestaus*

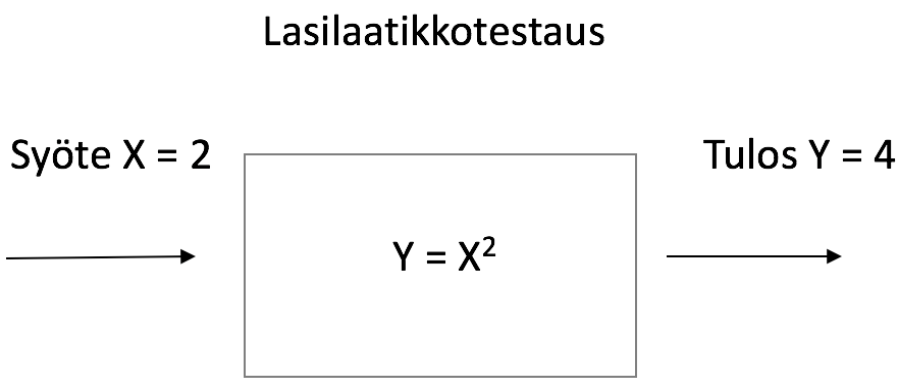
Mustalaatikkotestauksessa testitapauksien luonnissa on useita eri menetelmiä, joita ovat aluetestaus, ekvivalenttiositus, raja-arvoanalyysi ja virheenarvaus. Aluetestauksessa testataan kaikilla syöteavaruuden alueen arvoilla, kun taas ekvivalenttiosituksessa syötteet jaetaan ekvivalenssiluokkiin. Jos sovellus toimii yhdellä luokalla, niin se voidaan olettaa

toimivan myös muilla luokilla. Ekvivalenssiluokkia voivat olla esimerkiksi epäkelvolliset syötteet, liian pienet tai suuret syötteet ja kelvolliset syötteet. Raja-arvoanalyysissä annetaan testitapauksen syötteeksi luokkien rajoilla olevia arvoja tai syötteen luonteen mukaisia rajatapauksia. Virheenarvaus menetelmää käytetään sellaisissa tapauksissa, joissa epäillään syntyvän virheitä (Haikala & Märijärvi, 2006).

2.4.2 Lasilaatikkotestaus

Lasilaatikkotestaus on testaustapa, jossa nimenmukaisesti testaaja tietää sovelluksen sisäisen rakenteen. Testaustapaa kutsutaan myös rakenteelliseksi –tai valkolaatikkotestaukseksi (Haikala & Märijärvi, 2006).

Testauksessa hyödynnetään sovelluksen rakennetta testitapauksia laatiessa. Testitapauksiensuunnittelussa keskitytään sovelluksen ongelmakohtiin, jolloin se vaatii testaajalta tietämystä sen toteutuksesta ja toimintaperiaatteista sekä ohjelmointiosaamista. Ohjelmointitaustaisen testaajan on helpompi tunnistaa ohjelmapolut koodista ja näin ollen muodostaa niistä testitapauksia. Kuvan 5 mukaisesti testaaja valitsee testitapauksiin syötteitä ja päättelee sovelluksen rakenteen avulla syötteiden tulokset (Haikala & Märijärvi, 2006).



Kuva 5. *Lasilaatikkotestaus*

Lasilaatikkotestauksen huonoja puolia ovat hitaus ja sen takia tämä testausmenetelmä on hyvin kallis. Tämän lisäksi testaajalta vaaditaan hyvää osaamista, joka taitaa kyseisen ympäristön, koodikielen, testauksen ja ohjelmoinnin.

3. REGRESSIOTESTAUS

Regressiotestauksella tarkoitetaan aiemmin testatun sovelluksen uudelleen testausta ja se toteutetaan järjestelmätestausvaiheessa (Haikala & Märijärvi, 2006). Testauksen tavoitteena on varmistaa, että sovellus toimii uusien muutoksien jälkeen oikein, eikä aiheuta virheitä vanhemman testatun version toiminallisuuksiin. Uudet muutokset voivat olla esimerkiksi koodin poistamista, muokkaamista tai lisäämistä. Tamresin mukaan (Tamres, 2002) paras testiympäristö on sellainen, jossa regressiotestitapaukset suoritetaan joka kerta, kun sovellukseen on tehty muutoksia.

Sovellusta sanotaan regressoituneeksi, kun vanhasta testatusta toiminallisuudesta löytyy siihen tehdyn uusien muutoksien jälkeisessä regressiotestauksessa virhe. Regressiotestauksessa perustason versio (engl. baseline version) on komponentti tai systeemi, joka on läpäissyt testijoukon, kun taas systeemi, joka ei läpäissyt testijoukkoa kutsutaan deltaversioksi (engl. delta version). Näistä kummastakin versiosta ajettavaa koontiversiota kutsutaan deltakoosteeksi (engl. delta build) (Binder, 1999).

Regressiotestitapauksiksi (engl. regression test case) kutsutaan tapauksia, jotka testaavat vanhaa toiminallisuutta ja niitä käytetään myös deltakoosteiden testaukseen. Regressiotestaustavat on läpäisseet vanhan toiminallisuuden, joten niiden tulee läpäistä myös deltakoosteet. Jos regressiotestauksessa paljastuu uusia virheitä, niitä kutsutaan regressiovirheiksi (engl. regression fault) (Binder, 1999).

Regressiotestauksessa on kaksi vaihtoehtoa, joko testataan kaikki testitapaukset tai testataan osa testitapauksista (Rothermel & Harrold, 1996). Sovelluksen kasvaessa regressiotestauksesta tulee jokaisen uuden muutoksen jälkeen hyvin raskasta, jolloin on hyvä miettiä kannattaako kaikki testitapauksia ajaa. Regressiotestauksessa on hyvä löytää joukko testitapauksia, jotka testaavat sovelluksen kriittisimmät toiminallisuudet ja ne tapaukset, joissa on löytynyt eniten virheitä (Harrold, M. J., Gupta, R. & Soffa, 1993).

Testitapauksien valintaan on olemassa useita eri tekniikoita, kuten muutosperusteinen valintatekniikka (Wong, Horgan, London, Agrawal, & Street, 1997), jossa testitapaukseksi valitaan mahdollisimman tehokas joukko. Valintakriteereinä voidaan käyttää muutosperusteisessa tekniikassa minimisaatiota tai priorisointia. Minimisaatiolla tarkoitetaan mahdollisimman pientä joukkoa, jolla toiminallisuus pystytään varmistamaan. Priorisoinnissa taas ajetaan testitapauksia mahdollisimman paljon aloittaen tärkeimmästä toiminallisuudesta. Muutosperusteisen valintatekniikan lisäksi testitapaukset voidaan valita kattavuuden perustuvien (Rothermel & Harrold, 1996) tai turvallisen menetelmien mukaan. Turvallisessa menetelmässä valitaan kaikki sellaiset testitapaukset, jotka voivat löytää virheen sovelluksessa, kun taas kattavuuteen perustuvassa menetelmässä valitaan ainoastaan sellaiset testitapaukset, jotka liittyvät muuttuvaan osaan.

Regressiotestaus voidaan jakaa kahteen erilaiseen testaukseen (Leung & White, 1991), jotka ovat progressiivinen ja korjaava testaus. Progressiivisessa testauksessa testataan muuttunutta sovellusta alkuperäisten testitapauksien kanssa. Korjaavassa testauksessa testitapauksiin joudutaan tekemään muutoksia, koska uusien muutoksien lisäksi myös sovelluksen vaatimukset tai määritykset ovat muuttuneet, jolloin ne tulee testata uusien määrityksien mukaisesti.

Regressiotestaus on samojen testitapauksien toistamista, mikä olisi hyvin raskasta ja kallista manuaalisesti testattuna (Haikala & Märijärvi, 2006). Tämän takia se sopii hyvin automatisoinnin kohteeksi. Testitapauksien suorittamisen lisäksi voidaan automatisoida testitapausten muokkaaminen ja luominen, koska niitä tulee myös toistettua useasti uusien muutoksien ja määritysten takia. Jokaisen muutoksen jälkeen testit tulee ajaa joka kerta uudelleen, jotta uusista muutoksista voidaan löytää mahdolliset virheet. Tästä syystä automatisoiduista testeistä on hyötyä, koska testien ajokertoja on paljon.

4. TESTIAUTOMAATIO

4.1 Testauksen automatisointi

Lyhyesti sanottuna testauksen automatisointi on manuaalisen testauksen suorittamista koneellisesti jonkin sovelluksen avulla. Testiautomaatiossa annetaan koneelle käskyjä testitapauksien mukaan, jolloin se tekee saman asian kuin manuaalinen testaaja. On kuitenkin muistettava, että testiautomaatio ei korvaa manuaalista testausta, mutta vapauttaa testaajien aikaa.

Mittalin ja Acharyan (Mittal, N. & Acharya, 2003) mukaan on hyvä automatisoida testitapaukset, jotka vievät eniten aikaa ja resursseja testaukselta. Esimerkiksi jos ohjelmistoprojektiin kuuluu ohjelmiston ylläpitotyöt, kuten muutokset, päivitykset tai korjaukset, silloin kannattaa regressiotestaus automatisoida.

Ennen automatisointia pitää löytää toimiva manuaalinen testausjärjestelmä. Toimivan manuaalisen järjestelmän tulee sisältää seuraavat asiat: Yksityiskohtaiset testitapaukset ja niiden odotetut tulokset, jotka perustuvat suunnittelu- ja määrittelydokumentteihin. Tämän lisäksi tulee olla erillinen testiympäristö, jossa testitietokanta voidaan uudelleen tallentaa vakiomuodossa, jotta testitapaukset pystytään suorittamaan uudelleen jokaisen muutoksen jälkeen (Zambelich, 1998).

Testauksen automatisointiin liittyy useita perusteluita, kuten koneen ja ihmisen erot. Esimerkiksi kone voi havaita asioita, jotka ihmiseltä jäisi kokonaan huomaamatta tai se veisi paljon aikaa. Kone havaitsee esimerkiksi pitkät desimaaliluvut helposti, mutta ihmisille tulee helposti lyöntivirheitä pitkien lukujen kanssa. Tämän lisäksi ihminen tekee helposti testatessa virheitä ja jättää havaitsematta ne, kun taas kone löytää virheet suurista joukoista. Ihminen voi myös havaita ongelmia, joita kone ei havaitse. Tällaisia ovat esimerkiksi käytettävyysongelmat. Koneelle ei ole merkitystä, miten ja missä järjestyksessä nappulat ja dialogit ovat, mutta ihmisille on. Kone suorittaa kaikki automaatiotestit aina samalla tavalla, kun taas ihminen ei välttämättä näin teen, jolloin poiketessa testitapauksista voidaan löytää virheitä, joita kone ei havaitse (Marick, 2000).

Manuaalitestauksen ongelmat liittyvät testauksessa löytyneiden virheiden toistamiseen ja testitapauksen muistiin kirjoittamiseen. Manuaalitestausta suorittaessa usein jätetään testin vaiheet kirjottamatta ylös, jolloin virheen havaitessa sen toistaminen tulee olemaan hankalaa. Automaattitestejä ajaessa taas testien vaiheet jäävät lokeihin ylös ja virheet pystytään toistamaan, koska testit suoritetaan aina samalla tavalla. Manuaalitestauksen kulun järjestelmällinen ylös kirjaamisen jättäminen ja yleinen epäjärjestelmällisyys ovat Mittelin ja Archaryan mielestä yksi manuaalitestauksen ongelmista (Mittal, N. & Acharya, 2003).

Testauksen automatisointi ei ole aina kuitenkaan järkevin vaihtoehto, koska se on kalliimpaa kuin saman testitapauksen manuaalinen testaaminen. Gerrardin mielestä (Gerrard, 1997) regressiotestien automatisointi on kuitenkin järkevää, koska testitapaukset toistetaan joka kerta uudelleen muutoksien jälkeen. Gerrard lisää myös, että regressiotestien automatisointi skripteistä kannattaa tehdä sellaisia, että niillä voidaan löytää virheitä jo ohjelmistoprojektin alkuvaiheessa, koska silloin niiden hyötyä voidaan korostaa. Pitää kuitenkin muistaa, että regressiotestien tulee olla valideja sovelluksen valmistumisen jälkeen.

Automatisoitavat regressiotestit voivat olla esimerkiksi savutestejä (engl. smoke test), joilla varmistetaan sovelluksen kriittisimmät kohdat. Nämä ajetaan päivittäin jokaisen muutoksen jälkeen. Virheiden löytäminen on nopeaa, koska tiedetään, minkä muutoksen mukana virheet ilmenivät.

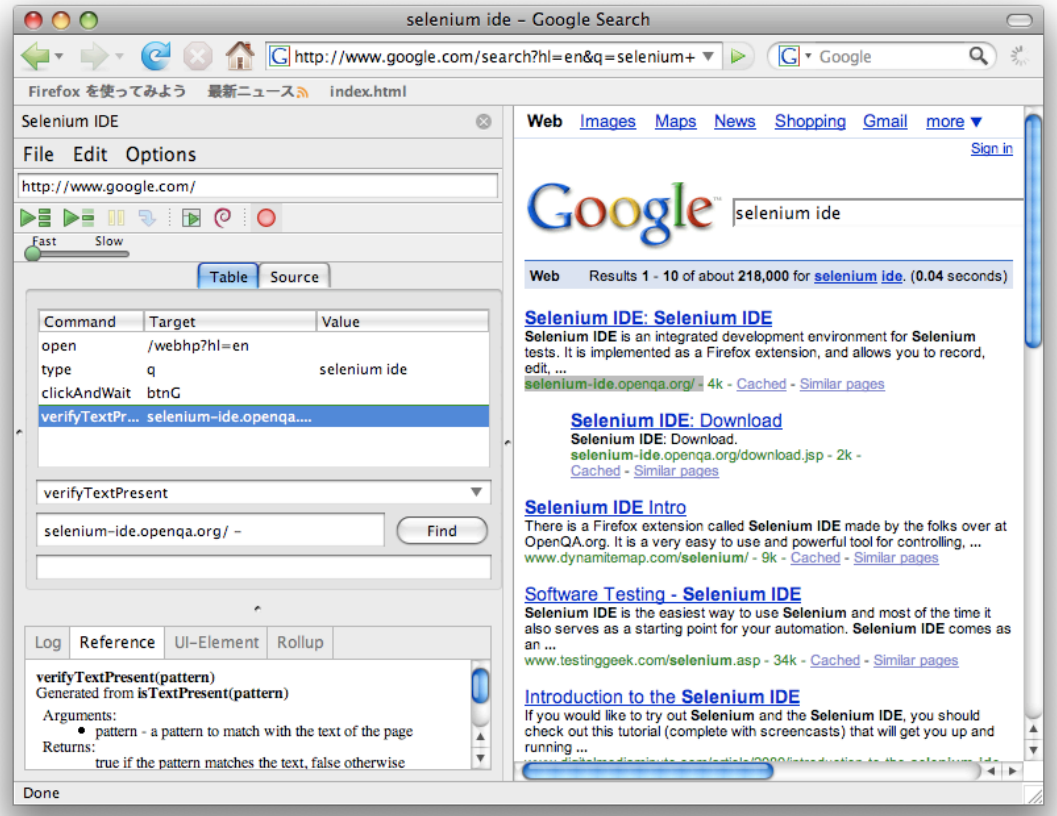
4.2 Automaatiotestauksen työkaluja Web-testauksessa

Web-testauksen automatisointiin on olemassa useita eri työkaluja. Taulukkoon 1 on koottu muutama työkaluista, jotka esittelen työssäni tarkemmin. Robot Framework ja Selenium ovat työkaluja, joita tulen käyttämään tässä työssä. Päätin tämän lisäksi mielenkiinnosta tutustua myös pariin muuhun työkaluun. Luvussa 5 tutustutaan Robot Frameworkiin ja sen asennukseen tarkemmin.

Taulukko 1. *Web-testauksen työkaluja*

Nimi	Järjestelmän vaatimukset	Mihin testaukseen sopiva?	Yritys	Automatisointi	Graafinen käyttöliittymätestaus
Robot Framework	Mikä tahansa	Kaikkiin	Avoin lähdekoodi	Kyllä	Kyllä
Selenium	Mikä tahansa	Web	Avoin lähdekoodi	Kyllä	Kyllä
Sahi	Mikä tahansa	Web	Avoin lähdekoodi	Kyllä	Kyllä
TestComplete	Windows	Web, Mobiili	SmartBear Software	Kyllä	Kyllä

Selenium on regressiotestaukseen sopiva avoimen lähdekoodin automaatiotestaus-työkalu, joka tukee Windows, Linux ja Macintosh ympäristöjä. Jason Huggins kehitti alun perin sen vuonna 2004 ThoughtWorksin sisäiseksi työkaluksi, mutta jo samana vuonna se oli avoinna käytössä kaikille (Selenium HQ, 2016).

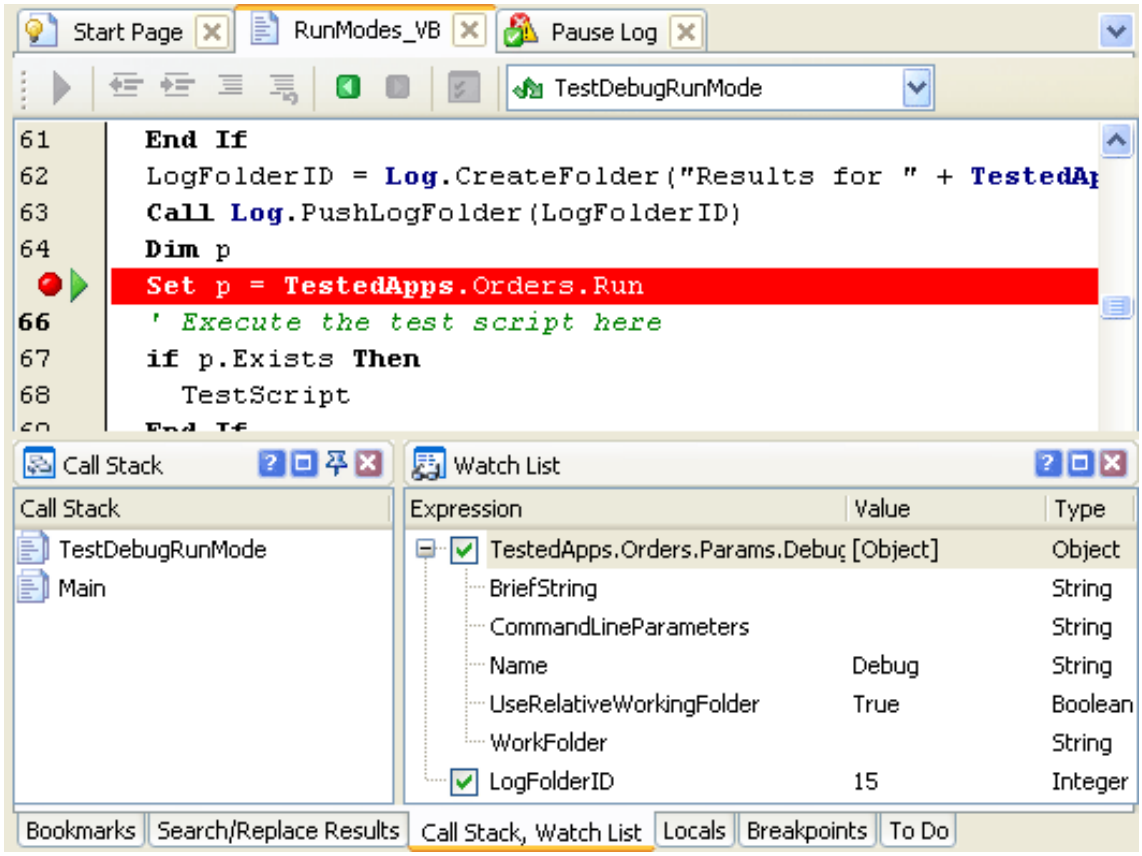


Kuva 6. *Selenium IDE:n käyttöliittymä* (Selenium HQ, 2016)

Selenium tarjoaa käytettäväksi Selenium IDE:ä (Kuva 6) laajennusta, joka on tallenna-ja-toista-työkalu. Selenium IDEä pystyy käyttämään ilman testaus scriptien osaamista. Tämän lisäksi Selenium tukee useita eri tunnettuja koodikieliä, joilla testaus scriptejä voi kirjoittaa, kuten esimerkiksi Java, C#, Python, PHP ja Ruby (Selenium HQ, 2016).

Sahi on web-sovelluksiin soveltuva työkalu, jota on tarjoilla ilmaisena sekä maksullisena versiona. Ilmainen versio sisältää testauksen olennaisimmat ominaisuudet, mutta kaupallisessa versiossa pystyy esimerkiksi räätälöimään ja tallentamaan raportteja tietokantaan sekä ottamaan kuvankaappauksia testien aikana. Sahi on kirjoitettu Javalla ja JavaScriptillä, mikä mahdollistaa testauksen tallentaminen ajon aikana (Sahipro, 2016).

TestComplete (kuva 7) on SmartBear Softwaren kehittämä funktionaalinen testausalusta, joka soveltuu Web-sovellusten ja mobiiliohjelmien testaukseen sekä sovelluksen yksikkötestaukseen (engl. unit testing).



Kuva 7. *TestCompele editori* (Smartbear, 2016)

Automaattitestejä pystytään ajamaan skriptien, tallenna-ja-toista-työkalun tai manuaalisesti avainkoodi operaatioiden avulla. Tämän lisäksi TestComplete soveltuu myös backend-testauksiin, joita on esimerkiksi tietokannan testaus (Smartbear, 2016).

Yhteenveto

Automaatiotestiin käytettäviä työkaluja on markkinoilla olemassa kymmeniä, joista siis vain murto-osa esitettiin kohdassa 4.2. Kaikki kohdassa 4.2 mainitut työkalut soveltuvat web-sovelluksen testaukseen, mutta kaikista ei ole saatavilla täysin ilmaista versioita.

Robot Framework on täysin ilmaiseksi tarjolla kaikille ja on yksi suosituimmista automaatiotestauksen työkaluista. Selenium IDE on taas helppokäyttöisin, koska testejä pystytään luomaan ilman skriptejä, mutta toimii ainoastaan Mozilla-selaimella. Sahi käyttää testien kirjoittamiseen skriptikieltä JavaScript, mutta vaatii maksullisen version lisäominaisuuksia varten. TestComplete sopii käytettäväksi web-sovelluksien lisäksi mobiilisovelluksien testaukseen, mutta vaatii myös maksullisen lisenssin.

Kohdassa 4.2 esitetyistä työkaluista valittiin käytettäväksi Robot Framework ja Seleniumin tarjoamat kirjastot, koska olivat testaajalle ennestään tuttuja. Tästä syystä uusien työkalujen perehdyttämiseen ei tarvinnut kuluttaa aikaa. Tämän lisäksi Robot Framework

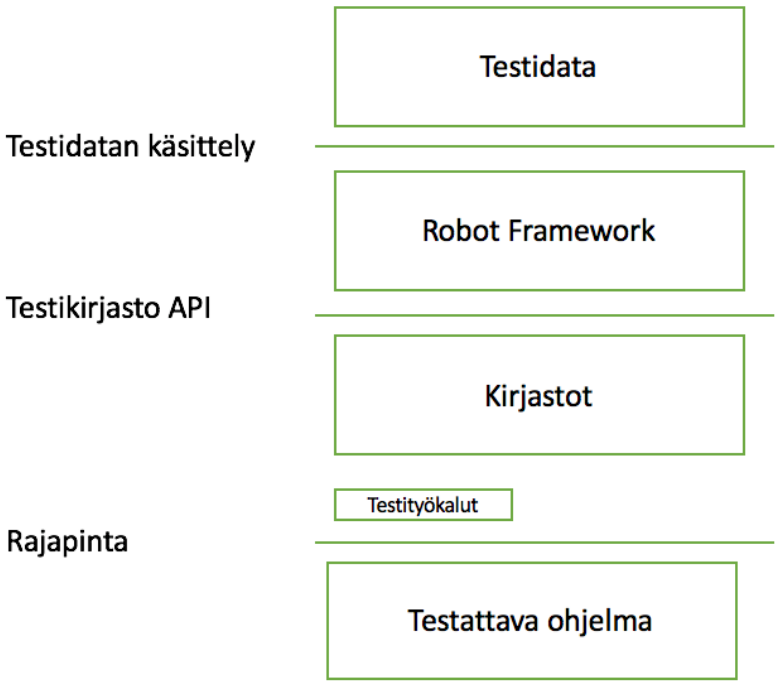
soveltuu erinomaisesti web-soveluksien testaamiseen, koska pystyy hyödyntämään valmiita Seleniumin tarjoamia kirjastoja.

5. ROBOT FRAMEWORK

5.1 Yleisesti Robot Frameworkista

Robot Framework on työkalu automaatiotestaukseen, jonka on kehittänyt Pekka Klärck (Klärck, 2009) vuonna 2005 Network Siemensillä. Vuonna 2008 siitä (Framework, 2015) julkaistiin avoimen lähdekoodin versio, joka on saatavilla kaikille. Robot Framework on Python pohjainen testauskehys, joka on laitteisto- ja teknologiariippumaton. Se sopii hyvin käytettäväksi hyväksymistestaukseen.

Robot Framework käyttää avainkoodityyppistä lähestymistapaa, jossa testit suoritetaan testitapauksissa esitetyn datan ja kirjastojen avulla. Kirjastot toimivat rajapintana testiympäristön (‚joka koostuu testidatasta ja Robot Frameworkista) ja testattavan järjestelmän (SUT) välillä (Framework, 2015).



Kuva 8. *Robot Framewok:n arkkitehtuuri*

Kun Robot Framework käynnistetään, se aluksi käsittelee testidatan, josta poimii käytettävät avainsanat. Avainsanojen avulla Robot Framework etsii kirjastoista käytettävät komennot, joiden avulla se testaa halutut toiminnot ja hyödyntää tarvittaessa ulkopuolisia

testityökaluja. Tämän jälkeen se vertaa testissä saatuja tuloksia testidatassa oleviin oletettuihin tuloksiin ja luo näistä HTML- ja XML-dokumentit. Kuvassa 8 on esitetty Robot Frameworkin arkkitehtuuri ja sen ympärillä käytettävät moduulit (Framework, 2015).

5.2 Testidatan tiedostoformaatit

Robot Framework tukee neljää eriä tiedostoformaattia, joita ovat HTML, TSV, reST ja puhdas tekstitiedostoformaatti (Nokia Solution and Networks, 2015).

HTML-tiedostot tukevat formatointia ja vapaata tekstiä testitapaustaulukoissa. Tämä mahdollistaa sen, että voi lisätä lisätietoja testitapauksiin ja luomaan testitapaustiedostoja, jotka näyttävät samalaiselta kuin määritellyt testitapaukset. HTML-formaatin huonoja puolia ovat sen vaikea muokkaaminen tavallisella tekstieditorilla sekä testitiedostojen versioiden vertailu versionhallinnassa (Nokia Solution and Networks, 2015).

Kuvassa 9 on esimerkki HTML-formaatista, jossa testidatat on eroteltu eri tauluihin, joita ovat asetus- (engl. Setting), muuttuja- (engl. Variable), testitapaus- (engl. Test Case) ja avainsana- (engl. Keyword) taulukot (Nokia Solution and Networks, 2015).

Setting	Value	Value	Value
Library	OperatingSystem		

Variable	Value	Value	Value
${MESSAGE}	Hello, world!		

Test Case	Action	Argument	Argument
My Test	[Documentation]	Example test	
	Log	${MESSAGE}	
	My Keyword	/tmp	
Another Test	Should Be Equal	${MESSAGE}	Hello, world!

Keyword	Action	Argument	Argument
My Keyword	[Arguments]	${path}	
	Directory Should Exist	${path}	

Kuva 9. *Kuvankaappaus HTML-formaatista* (Nokia Solution and Networks, 2015)

TSV-formaatissa kaikki testidata on määritelty yhdessä taulussa, jossa testidata on eroteltu toisistaan sarkaimien avulla. Kuvan 10 mukaisesti testitapaukset ja otsikot on eroteltu toisistaan asteriksien (*) avulla ja solut tabulaattorin avulla. TSV-tiedostoa pystyy luomaan ja muokkaamaan millä tahansa taulukkolaskentaohjelmalla ja TSV-formaatin versioita on myös helppo verrata versionhallinnassa (Nokia Solution and Networks, 2015).

Setting	*Value*	*Value*	*Value*
Library	OperatingSystem		
Variable	*Value*	*Value*	*Value*
${MESSAGE}	Hello, world!		
Test Case	*Action*	*Argument*	*Argument*
My Test	[Documentation]	Example test	
	Log	${MESSAGE}	
	My Keyword	/tmp	
Another Test	Should Be Equal	${MESSAGE}	Hello, world!
Keyword	*Action*	*Argument*	*Argument*
My Keyword	[Arguments]	${path}	
	Directory Should Exist	${path}	

Kuva 10. *Kuvankaappaus TSV-formaatista* (Nokia Solution and Networks, 2015)

reST-formaatti on helppolukuinen formaatti, jossa testidata on eroteltu toisistaan yhtäsuuruusmerkkien ja välilyöntien avulla. Kuvasta 11 voidaan huomata, että kaikki erotinmerkkien ulkopuolella olevaa teksti jätetään huomioimatta testauksessa.

```
This text is outside tables and thus ignored.

===========  ===============  =======  =======
  Setting         Value         Value    Value
===========  ===============  =======  =======
Library      OperatingSystem
===========  ===============  =======  =======


===========  ===============  =======  =======
  Variable        Value         Value    Value
===========  ===============  =======  =======
${MESSAGE}   Hello, world!
===========  ===============  =======  =======


=============  ================  ============  =============
  Test Case         Action          Argument      Argument
=============  ================  ============  =============
My Test        [Documentation]   Example test
\              Log               ${MESSAGE}
\              My Keyword        /tmp
\
Another Test   Should Be Equal   ${MESSAGE}    Hello, world!
=============  ================  ============  =============
```

Kuva 11. *Kuvankaappaus reST-formaatista* (Nokia Solution and Networks, 2015).

Puhtaassa tekstitietiedostoformaatissa voi käyttää mitä tahansa tekstieditoria. Sen hyviä puolia ovat sen helppo muokattavuus ja versioiden vertaileminen versionhallinassa. Puhdas tekstitiedosto ja TSV-formaatti ovat hyvin samankaltaiset, mutta solujen erottimet

ovat hieman erilaiset. Puhtaassa tekstitiedostoformaatissa erottimena käytetään kahta tai useampaa välilyöntiä tai putkimerkkejä. Robot Framework kuitenkin suosittelee käyttämään neljää välilyöntiä erottimena. Putkimerkki on näistä selkeämpi ja luettavampi, mutta välilyöntejä on helpompi käyttää (Nokia Solution and Networks, 2015).

Puhtaassa tekstitiedostoformaatissa on hyvä käyttää Robot Framewor syntaksia tukevaa editoria, koska ne helpottavat testitapausten kirjoittamista ja luettavuutta. Tämän lisäksi editorit ehdottavat testiä kirjoittaessa olemassa olevia avainsanoja, testitapauksia jne. Kuvasta 12 voidaan huomata, että testitapaukset (valkoinen), toimenpiteet (sininen) ja argumentit (oranssi) on eroteltu toisistaan eri väreillä (Nokia Solution and Networks, 2015).

```
*** Settings ***
Documentation     A test suite with a single test for valid login.
...
...               This test has a workflow that is created using keywords in
...               the imported resource file.
Resource          resource.txt

*** Test Cases ***
Valid Login
    Open Browser To Login Page
    Input Username    demo
    Input Password    mode
    Submit Credentials
    Welcome Page Should Be Open
    [Teardown]    Close Browser
```

Kuva 12. *Kuvankaappaus puhtaasta tekstitiedostoformaatista* (Framework, 2015)

Robot Frameworkilla (Nokia Solution and Networks, 2015) on olemassa useita sääntöjä, jolloin testidata jätetään huomioimatta testauksesta:

- Kaikki taulukot, joiden ensimmäisessä solussa ei ole Robot Framework tunnistettua taulukkonimeä, joita ovat Settings, Variable, Test Case ja Keyword.
- Kaikki tyhjät rivit.
- Kaikki tyhjät solut rivien lopussa, ellei rivejä oli päätetty lopetus-merkkiin (engl. escape character).
- Jos rivin ensimmäisessä solussa ensimmäinen merkki on risuaita. Muutoin se on kommentti.

Testidatan huomioimatta jättämistä käytetään paljon parantamaan testitapauksen luettavuutta, väliaikaiseen kommentointiin tai tyhjän arvon käyttämistä argumenttina avainsanoissa.

5.3 Testiautomaatiotyökalujen asennus

Diplomityössä asennettiin Robot Framework OS X –käyttöjärjestelmään, koska testaajalla oli kokemusta Linux- ja OS X-ympäristöissä. Robot Frameworkin lisäksi tuli asentaa tietokoneelle Python, Pip, Selenium2Library ja PyCharm-kehitystyökalu.

5.3.1 Python 2.7

Asennus aloitettiin Python 2.7 asennuksella, koska Robot Framework tarvitsee Pythonia tai Jythonia toimiakseen. Tietokoneelle asennettiin Python 2.7.1 version, koska Robot Framework 2.9 version kanssa tulee käyttää vähintään Pythonista versiota 2.7. Aikaisemmat versiot Robot Frameworkista tukevat Python 2.3 versiota. Nykyään Robot Frameworkin uusi versio 3.0 tukee Pythonin 3.0 versiota (Santos, 2009).

Ennen Pythonin asennusta tarkistettiin terminaalista komennolla ”python -V”, että onko tietokoneeseen jo asennettu Pythonia. Python 2.6 löytyi esiasennettuna OS X-käyttöjärjestelmästä, mutta haluttiin asentaa Pythonista uusin versio ja samalla tutustua OS X-käyttöjärjestelmän omaan paketinhallintajärjestelmään (engl. package manager) homebrew:iin. Paketinhallintajärjestelmän avulla pystyy helposti asentamaan, päivittämään ja poistamaan tietokoneohjelmia käyttöjärjestelmästä.

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/in-
stall/master/install)"
brew help
echo "export PATH=/usr/local/bin:/usr/local/sbin:$PATH" &gt;&gt;
~/.profile
source ~/.profile
```

Yllä olevien komentojen avulla saatiin asennettua koneelle homebrew ja lisättiin hombrew ympäristömuuttujaan ”Path”. Tämän jälkeen asennettiin homebrew:in avulla Python tietokoneelle seuraavalla komennolla:

```
brew install python
```

Ennen homebrewin asennusta yritettiin Robot Frameworkin 2.9 versiota ajaa Python 2.6 version kanssa ilman onnistumista. Myöhemmin vasta havaittiin, että Robot Frameworkin 2.9 versio tarvitsee toimiakseen Pythonin 2.7 version. Tämän jälkeen yritettiin päivittää Python ilman paketinhallintajärjestelmää, mutta se koitui hankalaksi, joten siirryttiin usean epäonnistuneen kokeilun jälkeen asentamaan homebrew paketinhallintajärjestelmä. Python päivityksen jälkeen siirryttiin Robot Frameworkin ja Selenium2Libraryn asennukseen.

5.3.2 Robot Frameworkin ja Selenium2Library:n asennus

Ennen Robot Frameworkin asennusta asennettiin tietokeelle Pythonin paketinhallintajärjestelmä pip, jonka kautta Robot Frameworkin asennus onnistuu helposti. Pythonin versioon 2.7.1 ei vielä Pip sisälly, joten se piti ensin ladata internetistä ja asentaa koneelle komennolla:

```
python get-pip.py
```

Tämän jälkeen Robot Frameworkin (Santos, 2009) uusimman version asennus onnistui Pip:n avulla seuraavalla komennolla:

```
pip install robotframework
```

Yllä olevan komennon avulla olisi myös pystynyt asentamaan halutun version laittamalla ”robotframework”-komennon jälkeen ”=<versionumero>”. Pip:n avulla pystyy myös päivittämään Robot frameworkia ”—upgrade” option avulla sekä poistamaan asennuksen uninstall-komennon avulla. Tietokoneella asennettiin kuitenkin uusin versio Robot Frameworkista ja sen jälkeen alla olevan komennon avulla pystyi varmistamaan, että Robot Framework on asennettu oikein (Santos, 2009):

```
pybot --version
Robot Framework 2.9.2 (Python 2.7.10 on darwin)
```

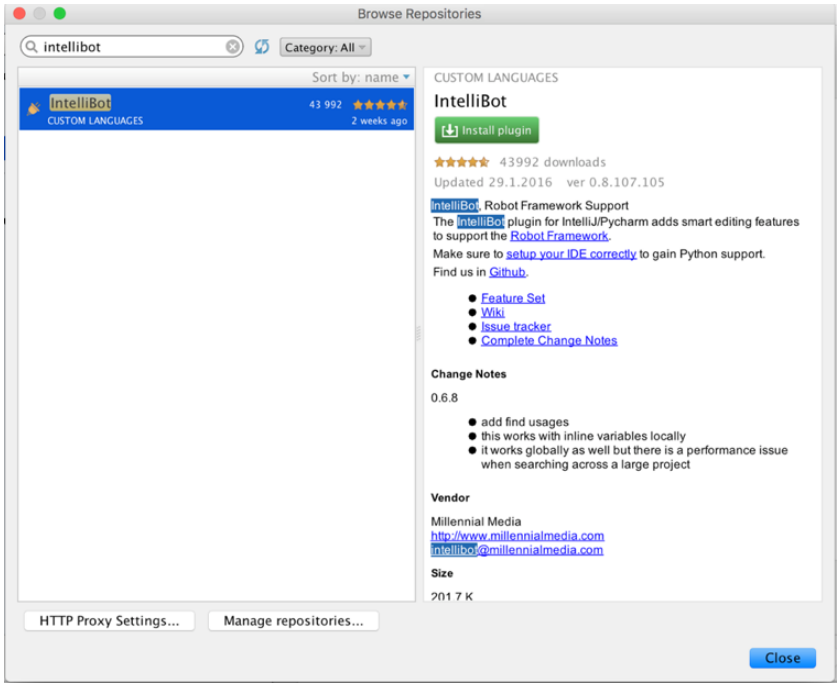
Robot Framework asennuksen jälkeen siirryttiin asentamaan Selenium2Library:a (Santos, 2009), joka on web-testauskirjasto. Sen avulla pystyy helposti kirjoittamaan testejä Web-ohjelmistoille. Selenium2Libraryn asennus Robot Frameworkiin tapahtui samankaltaisesti kuin Robot Frameworkin asennus:

```
pip install robotframework-selenium2library
```

Näiden asennuksien jälkeen siirryttiin asentamaan PyCharm-kehitystyökalua, jonka avulla pystytään kirjoittamaan automaatiotestejä.

5.3.3 PyCharm-kehitystyökalun asennus

PyCharm on kehitystyökalu Python-ohjelmointiin (Taft, 2010), jonka on kehittänyt tšekkiläinen yritys JetBrains. PyCharmiin on mahdollista asentaa Robot Framework-liitäntä, minkä perusteella päädyttiin asentamaan kyseinen kehitystyökalu.



Kuva 13. *PyCharm IntelliBot-liitännän asennus*

PyCharmin asennus oli yksinkertainen: asennuspaketti ladattiin internetistä ja asennettiin tietokoneelle. Tämän jälkeen käynnistettiin PyCharm-ohjelma ja siirryttiin ohjelman asetus-sivulle (PyCharm – Preferences – Plugins), josta pystyy asentamaan liitännäisiä. Hakulomakkeeseen kirjoitettiin ”inttelibot” (Kuva 13), josta sen pystyi helposti asentamaan PyCharmiin.

Tämän jälkeen PyCharm pyytää käynnistämään ohjelman uudelleen, jonka jälkeen se tukee Robot Framework -ympäristöä.

5.3.4 Yhteenveto

Robot Frameworkin asennus onnistui hyvin, koska ohjelman avulla pystyttiin käynnistämään testitapaus onnistuneesti sekä se loi testauksen jälkeen html-lokit testauksen kulusta. Tämän lisäksi ohjelma suoriutui tekemään epäonnistuneen testitapauksen jälkeen lokitiedostoihin kuvakaappaukset, joita Selenium2Library-kirjasto tuottaa.

Asennuksen aikana ongelmaksi koitui Python version 2.7 asennus ja sen lisääminen ympäristömuuttujaan ”Path”. Vaikka asennus suoritettiin ohjeiden mukaisesti, ei tietokone löytänyt Pythonista versiota 2.7 vaan alkuperäisen 2.6, joka oli valmiiksi asennettuna OS X-käyttöjärjestelmään. Internetistä löytyi kuitenkin käyttöohjeet OS X-käyttöjärjestelmän paketinhallintajärjestelmään homebrew, jonka avulla asennus sujui ongelmitta.

5.4 Testien kirjoittaminen

Diplomityössä testit kirjoitettiin Robot Frameworkilla, joka käytti testien suorittamiseen Selenium2Library-kirjastoa. Testien kirjoittamisessa kehitysympäristönä käytettiin PyCharmia, joka tukee Robot Framework testejä ja helpottaa niiden kirjoittamista ja ajamista. Automaatiotestien toteutus aloitettiin tutustumalla PyCharm-kehitysympäristöön jakamalla testit Robot Frameworkin käyttöohjeiden mukaisiin kokonaisuuksiin ja ajamalla testejä ohjelmiston testiympäristössä.

5.4.1 Testien jakaminen kokonaisuuksiin

Yleensä automaatiotestaukset testit jaetaan kokonaisuuksiin luomalla jokaiselle ajettavalla testille oma tiedostonsa, joita sitten ajetaan. Tässä tapauksessa kuitenkin ylläpidettävyys kärsii, jos halutaan käyttää samaa testitapausta useassa eri testitiedostossa. Robot Frameworkilla on olemassa helppo ja toimiva mekanismi tällaisten tapausten ratkaisussa.

Robot Frameworkissa (Nokia Solution and Networks, 2015) voidaan testitapauksille antaa merkintöjä (engl. tag) ja automaatiotestin käynnistäessä testaaja pystyy merkintöjen avulla valita mitä testejä halutaan ajaa. Robot Frameworkilla on useita eri käyttötarkoitusta mihin merkintöjä voidaan käyttää:

- Merkinnät näkyvät raporteissa ja lokeissa, jolloin ne antavat hyödyllistä dataa testauksesta.
- Statistiikkaa testauksesta, esimerkiksi lukumäärää onnistuneista ja epäonnistuneista testeistä.
- ”Include”- ja ”exclude”-parametrien avulla testaaja voi päättää mitä merkittyjä testejä ajetaan.
- Merkintöjen avulla testaaja pystyy määrittämään testauksien tasot, esimerkiksi mikä testi on kriittinen.

Merkinnän lisäksi on olemassa toinen tapa jakaa testit kokonaisuuksiin - luomalla testeistä testisarjoja (engl. test suites). Tällöin tarvitsee vain kutsua testisarjaa ajon aikana, jolloin se ajaa testisarjan alla olevat testitapaukset.

Testitiedostot voidaan jakaa myös kansioihin, joista Robot Framework testaa kaikki löytyvät testitiedostot. Käyttöohjeissa kerrotaan, että yhdessä testitiedostosta olisi hyvä olla enintään kymmenen testitapausta. Tämän lisäksi, jos halutaan käyttää kansioita testauksien jakamiseen, siihen on olemassa kolme ehtoa Robot Frameworkilta (Nokia Solution and Networks, 2015):

- Testiä ei suoriteta, jos kansion nimi alkaa pisteellä (.) tai alaviivalla (_)
- Testiä ei suoriteta, jos tiedoston nimi on ”CSV”
- Testiä ei suoriteta, jos tiedostolla ei ole jotain seuraavista tiedostopäätteistä: .html, .xhtml, .htm, .tsv, .txt, .rst, tai .rest, .robot

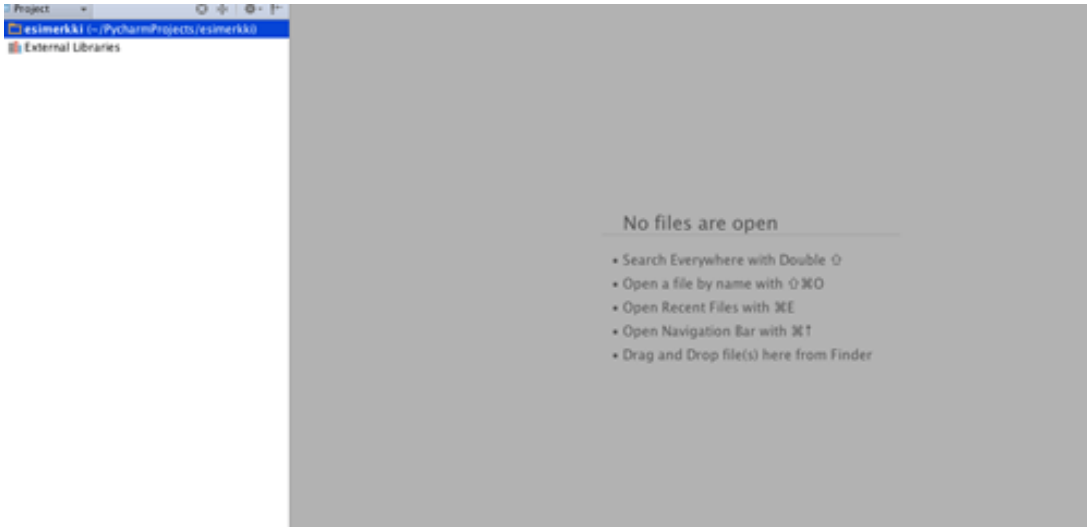
Jos testitiedosto tai kansio, jota ajetaan ei sisällä testitapauksia, sivuutetaan se ilman erillistä viestiä ja testaus jatkuu tämän jälkeen normaalisti. Ainoastaan Robot Frameworkin syslogiin tulee viesti sivuutetusta testistä.

5.4.2 PyCharm

Diplomityössä asennettiin PyCharm-kehitystyökalu testien kirjoittamista varten. Sen asennus on helppo ja käyttöliittymä oli ennestään tuttu testaajalle, joten sen takia päädyttiin käyttämään PyCharmia tässä työssä. Alakohdassa 5.5.3 asennettiin Robot Framework -liitäntä, joten ympäristö on valmis testitapauksien kirjoittamiselle.

PyCharmin käynnistäessä avautuu tyhjä ikkuna ja yläpuolella löytyvät valikot, josta File-valikon alta pystyy luomaan uuden projektin. Uuden projektin luomisen jälkeen PyCharm näyttää kuin kuvassa 14.

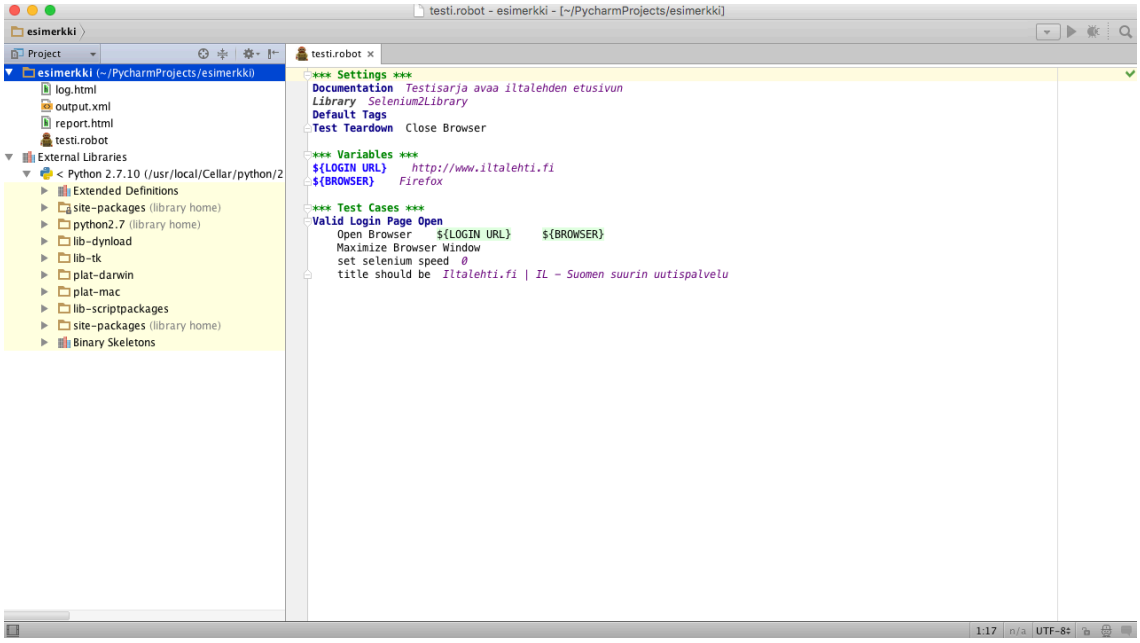
Kuvassa 14 avautuu tyhjä työtila, jossa vasemmalla puolella näkyy juuri luotu uusi testiprojekti nimeltä ”esimerkki”. Tämän alla näkyy käytettävissä olevat ulkopuoliset kirjastot. Työssä tullaan käyttämään Selenium2Library-kirjastoa, jonka avulla pystyy helposti kirjoittamaan testitapauksia web-sovelluksille.



Kuva 14. *PyCharm projektin luomisen jälkeen*

Projektin luomisen jälkeen aloitetaan luomaan testitiedostoja, jotka tukevat Robot Frameworkia. Tiedosto voidaan luoda painamalla projektin kohdalla hiiren vasenta näppäintä tai File-valikosta ”New” ja ”File”. Tämän jälkeen nimetään tiedosto ja annetaan tiedoston päätteeksi ”.robot”, jonka jälkeen se tukee Robot Framework syntaksia ja testit voidaan suorittaa terminaalin kautta.

Kuvassa 15 näkyy esimerkkitesti PyCharmilla tehtynä. Siinä näkyvät ”Settings”, ”Variables” ja ”Test Case” -testidatat.



Kuva 15. *Esimerkki automaatiotestistä PyCharmilla*

Asetukset (engl. Settings) kohdassa esitellään testisarja ja mitä kyseisessä testisarjassa tullaan testaamaan. Tämän jälkeen otetaan käyttöön Selenium2Libary – kirjasto ja oletus merkit (engl. default tags). Kohdassa asetukset voidaan myös ottaa käyttöön muita testisarjoja, avaintiedostoja, kirjastoja tai asetuksia, mitä ennen varsinaista testitapausta tai sen jälkeen tulee tehdä. Nämä ennen ja jälkeen asetukset saadaan käyttöön ”Test Setup”- ja ”Test Teardown” -parametreilla. Tässä tapauksessa varsinaisen testauksen jälkeen suljetaan kaikki testauksessa käytetyt selaimet ”Close Browser” -komennolla.

Muuttuja kohdassa ”Variables” voidaan asettaa muuttujia, joita käytetään testauksessa. Yleensä on suositeltavaa käyttää muuttujia, koska tällöin testitiedostot ovat paremmin ylläpidettäviä.

Testitapauskohdassa luodaan varsinainen testi. Kuvan 15 esimerkissä testataan, että avautuuko selain ja esimerkkisivu. Testaus varmistetaan onnistuneeksi antamalla sivun otsikko (engl. title) ”title should be” -komennon parametriksi.

5.4.3 Testien ajaminen

Testien ajaminen tapahtuu terminaalin kautta ”pybot”-komennon avulla. Alapuolella on esimerkki komento testitapauksen ajamisesta, jossa ”pybot”-komennon jälkeen kutsutaan testitiedostoa, joka halutaan ajaa:

```
pybot testi.robot
```

”Pybot”-komento toimii Robot Frameworkin 2.9: ssa ja sitä vanhemmissa versioissa ja sen jälkeisissä versioissa käytetään komentoa ”rebot”. Kun testi on suoritettu, antaa Robot

Frameworkin terminaali tiedon testin tuloksista kuvan 16 mukaisesti. Kuvasta voidaan huomata, että testisarja on suoritettu onnistuneesti ja testisarja sisälsi yhden testitapauksen.

```
==============================================================================
Testi :: Testisarja avaa iltalehden etusivun
==============================================================================
Valid Login Page Open                                                 | PASS |
------------------------------------------------------------------------------
Testi :: Testisarja avaa iltalehden etusivun                          | PASS |
1 critical test, 1 passed, 0 failed
1 test total, 1 passed, 0 failed
==============================================================================
Output:  /Users/veerushka/PycharmProjects/output.xml
Log:     /Users/veerushka/PycharmProjects/log.html
Report:  /Users/veerushka/PycharmProjects/report.html
```

Kuva 16. *Suoritetun testisarjan tulokset*

Tuloksien lisäksi Robot Framework lisää projektin juureen tuloste-, loki- ja raportti-tiedostot, joista näkee tarkemmin testisarjan testitapaukset ja niiden jokaisen vaiheen. Onnistuneessa testissä lokitiedosto on vihreän ja epäonnistuneessa punaisen värinen.

Testi Test Log

Generated
20160314 19:44:44 GMT +03:00
1 hour 9 minutes ago

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	1	0	00:00:15	
All Tests	1	1	0	00:00:15	

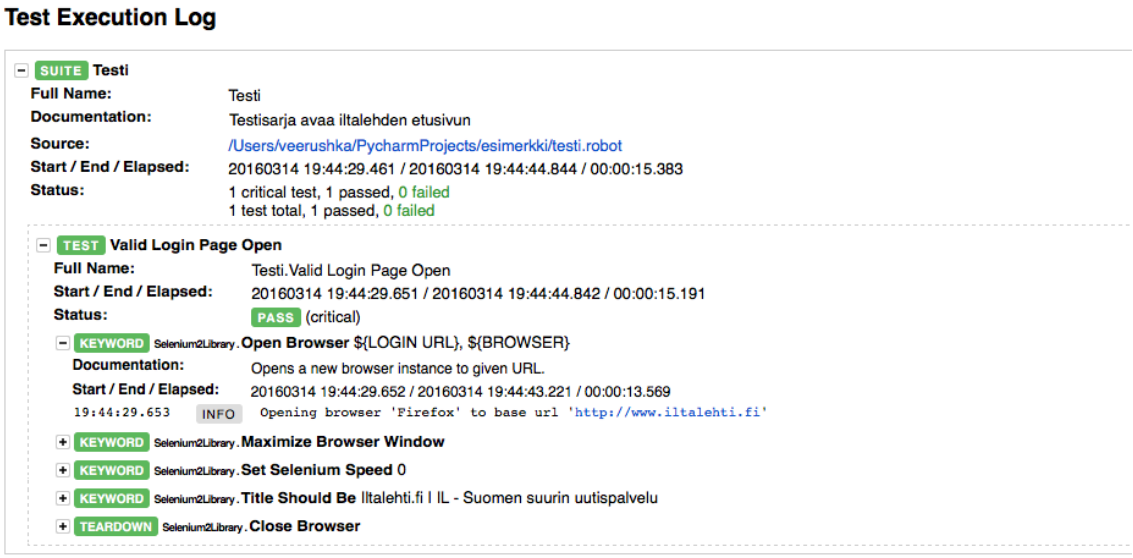
Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Testi	1	1	0	00:00:15	

Kuva 17. *Kuvankaappaus lokitiedoston yläosasta*

Kuvassa 17 on esitelty onnistuneen testisarjan lokitiedoston yläosa, josta näkee koko testisarjan suoritusajan, kaikkien testitapauksien yksittäiset suoritusajat sekä testien ja onnistuneiden testien määrän.

Kuvassa 18 on esitetty lokitiedoston alaosa, jossa on avattu testitapauksien kaikki vaiheet auki. Tämä helpottaa testien seuraamista ja pystyy varmistamaan, että kaikki tarpeelliset vaiheet tuli suoritettua. Lokitiedoston avulla pystyy myös helposti selvittämään missä kohtaan testi on epäonnistunut. Tämän lisäksi testauksesta ulkopuoliset henkilöt ymmärtävät testauksen kulkua helposti niiden avulla.



Kuva 18. *Kuvakaappaus lokitiedoston alaosa*

Lokitiedostosta alaosa näkee testitapauksissa käytetyt avainsanat ja mitä ne konkreettisesti tekemät. Tämän lisäksi näkee jokaisen avainsanan suoritusajan.

6. TESTAUKSEN SUUNNITTELU JA TOTEUTUS

6.1 Testattavat testitapaukset

Diplomityön tarkoituksena on automatisoida web-sovelluksen regressiotestaus, jotta sovelluksen päivityksien jälkeinen laadunvarmistus olisi nopeampaa. Regressiotestauksessa on tarkoituksena testata uutta versioita vanhoilla testitapauksilla, jotta voidaan varmistaa, että uudet muutokset eivät riko vanhaa toimivaa sovellusta.

Ennen tätä työtä regressiotestaus on tehty manuaalisesta, mutta on huomattu sen vievän paljon aikaa, ja näin ollen halutaan automatisoida useasti toistettavat testitapaukset. Testattava sovellus on käytössä oleva verkkokauppa, joten laadunvarmistus on yrityksen kannalta erittäin tärkeässä roolissa jokaisen päivityksen jälkeen.

Testattavat testitapaukset ovat tapauksia, jotka ovat päivittäisessä käytössä sovelluksessa, joten niiden avulla pystytään varmistamaan tarvittava laatu. Tässä työssä luodut testitapaukset ovat uuden asiakastilin rekisteröinti, kirjautuminen asiakas-sivulle, ostoksen tilaaminen ja navigointi sovelluksen jokaiselle sivulle. Tätä testausta voidaan kutsua savutestauksessa, joka on osa regressiotestausta. Siinä tarkoituksena on testata sovelluksen perusominaisuudet. Kohdissa 6.3 ja 6.4 esitellään uuden asiakkaan rekisteröinnin käyttöja testitapaus.

6.2 Testattava sovellus

Tässä työssä testattava sovellus on sauna- ja sisustuspuutavaran verkkokauppa. Verkkokaupassa on myytävänä mittatilaustyönä rakennettuja lauteita sekä sisustuspuutavaraa. Se on rakennettu OpenCart verkkokauppa -alustan päälle.

Verkkokauppasovellus on hyvin yksinkertainen. Käyttäjä pystyy rekisteröimään verkkokauppaan olemassa olevan sähköpostin avulla, jonka jälkeen ostoksia pystyy tekemään kirjatumalla omilla tunnuksilla. Verkkokaupasta voi tilata saunamateriaaleja, saunanlauteet mittatilaustyönä, listoja jne. Kun ostokset on siirretty ostoskoriin, siirrytään kassalle, josta sovellus laskee automaattisesti kuljetuksen hinnan osoitetietojen mukaan. Tämän jälkeen ostokset maksetaan valitun maksutavan mukaisesti.

6.3 Esimerkkikäyttötapaus

Testivetoisessa kehityksessä (TDD) ennen testitapauksien kirjoittamista tulee kirjoittaa ylös käyttötapaukset (engl. use cases) testattavasta ohjelmistosta. Käyttötapaukset kuvaavat sovelluksen ja käyttäjän välistä vuorovaikutusta sarjana toimintoja, joita käyttäjä suo-

rittaa saavuttaakseen jonkin tavoitteen (Luukkainen, 2009). Alla esitellään esimerkki-käyttötapaus diplomityössä testattavasta ohjelmistosta, jossa rekisteröidään sovellukseen uusi asiakastili:

- Käyttäjä: asiakas
- Tavoite: rekisteröidä uusi asiakastili
- Laukaisija: asiakkaan tarve
- Esiehto: ei ole esiehtoa
- Jälkiehto: uusi tili rekisteröity onnistuneesti
- Käyttötapauksen kulku:

  1. Asiakas navigoi etusivulta rekisteröinti-sivulle
  2. Sovellus pyytää täyttämään tiedot rekisteröintiä varten
  3. Asiakas täyttää henkilökohtaiset tiedot
  4. Asiakas täyttää osoitetiedot
  5. Asiakas täyttää salasanan
  6. Asiakas vahvistaa salasanan
  7. Sovellus pyytää hyväksymään ehdot
  8. Asiakas hyväksyy ehdot
  9. Asiakas luo uuden tilin painamalla ”Jatka”-nappulaa
  10. Asiakas varmistaa, että sovellus navigoi automaattisesti juuri luodulle asiakas-tili-sivulle

- Poikkeuksellinen toiminta:

  10a. Asiakas ei voi rekisteröidä uutta tiliä, jos sovelluksesta löytyy jo olemassa oleva tili rekisteröitävälle asiakkaalle

Yllä olevassa käyttötapauksessa esiehdolla tarkoitetaan tilannetta, joka pitää olla valmiina, jotta käyttötapaus voidaan käynnistää. Jälkiehto taas tarkoittaa tilannetta onnistuneen käyttötapauksen jälkeen. Poikkeuksellisessa toiminnassa on kirjattu kaikki toiminnot, mitä tapahtuu, jos käyttötapauksen kulku epäonnistuu.

Käyttötapauksen kulussa on kaksi mahdollista lopputulosta, jotka ovat onnistuminen tai epäonnistuminen. Edellytykset onnistuneeseen lopputulokseen määritellään seuraavasti:

- Testaaja pystyy navigoimaan etusivulta rekisteröinti-sivulle
- Testaaja pystyy täyttämään pakolliset tiedot rekisteröinti-sivulla
- Sovellus ilmoittaa testaajalle, jos tiedot ovat puutteelliset
- Testaaja pystyy rekisteröimään uuden tilin ja siirtymään asiakastili-sivulle

Edellytykset epäonnistuneeseen lopputulokseen määritellään taas seuraavasti:

- Testaaja ei pysty navigoimaan rekisteröinti-sivulle
- Testaaja saa virheilmoituksia täyttäessään tietoja rekisteröinti-sivulla
- Testaaja ei pysty rekisteröimään uutta tiliä, vaikka tiedot ovat oikein
- Sovellus ei siirry automaattisesti asiakastili-sivulle rekisteröinnin jälkeen

Yllä olevien ehtojen mukaan voidaan analysoida sovelluksen toimivuutta.

6.4 Esimerkkitestitapaus

Testitapauksen on kirjoitettu käyttötapauksen vaatimuksien mukaisesti. Niissä käytetään Robot Frameworkin puhdasta tekstitiedostoformaattia sekä hyödynnetään avainsanoja testien suorittamiseen. Kuvassa 19 on esimerkki testisarjasta, jossa testaaja rekisteröi uuden asiakastilin sovellukseen luvussa 6.3 esitetyn käyttötapauksen mukaisesti.

```
*** Settings ***
Documentation     Testisarja testaa uuden asiakastilin rekisteröintiä
...
...               This test has a workflow that is created using keywords in
...               the imported resource file.
Resource          resource.robot
Suite Teardown    Close All Browsers


*** Test Cases ***
Register A New Client
    Open Browser To Index Page
    Go To Register Page
    Add Valid Values
    Confirm Registeration
    page should contain     Tilisi on luotu!

Register Existing Client
    Open Browser To Index Page
    Go To Register Page
    Add Valid Values
    Confirm Registeration
    page should contain     Varoitus: Sähköpostiosoite on jo rekisteröity!


*** Keywords ***
Go To Register Page
    Click Link  Asiakastilini
    Click link  Rekisteröidy
    wait until page contains    Henkilökohtaiset tiedot

Add Valid Values
    Input Text  id=input-firstname    Matti
    Input Text  id=input-lastname  Testaaja
    Input Text  id=input-email      matti.testaaja6@outlook.com
    Input Text  id=input-telephone  025512112
    Input Text  id=input-address-1   testikatu 1
    Input Text  id=input-city   Pori
    Select From List By Label   id=input-zone    Länsi-Suomen lääni
    Input Text  id=input-password   testiasiakas345
    Input Text  id=input-confirm    testiasiakas345

Confirm Registeration
    select checkbox  agree
    Click Button   Jatka
```

Kuva 19. *Esimerkkitestitapaus*

Testisarja sisältää kaksi testitapausta, jotka ovat uuden asiakkaan ja olemassa olevan asiakkaan rekisteröinti. Ensimmäisessä testitapauksessa testataan käyttötapauksen kulkua ja toisessa käyttötapauksen poikkeustapausta.

Testitiedoston aluksi on kerrottu lyhyesti testattava tapaus ja sen jälkeen ”Resource”-komennon avulla haettu tiedostosta ”resource.robot” avainsanat, joita tarvitaan useammassa eri testitapauksessa. ”resource.robot”-tiedostossa on esimerkiksi asetettu Selenium2Library-kirjasto, testattavan sovelluksen URL, avattava selain ja asetettu testauksen viive. Kun kaikki testitapaukset on testattu, suljetaan selaimet ”Suite Teardown”-komennon avulla.

Molemmat testitapaukset käyttävät samoja avainsanoja, jotka on määritetty ”Keyword:n” alapuolella. Uuden asiakkaan rekisteröinnissä aluksi avataan selain ja navigoidaan testattavan sovelluksen etusivulle ”Open Browser To Index Page”-avainsanan avulla. Tämän jälkeen navigoidaan rekisteröintisivulle ”Go To Register Page”-avainsanalla, joka on määritetty kyseisessä testitiedostossa. Navigointien jälkeen lisätään validit arvot kenttiin ”Add Valid Values”-avainsanalla, jonka jälkeen vahvistetaan rekisteröinti avainsanalla ”Confirm Registeration”. Lopuksi varmistetaan, että sovellus automaattisesti navigoi asiakastili-sivulle rekisteröinnin jälkeen ”Page Should Contain”-komennolla.

Poikkeustapaus eli olemassa olevan asiakkaan rekisteröinnin testaus on hyvin samankaltainen kuin uuden asiakkaan rekisteröinti. Siinä tehdään samat toimenpiteet, mutta ainoastaan varmistetaan, että sovellus ei rekisteröi olemassa olevaa asiakasta. ”Page Should Contain”-komennon avulla voidaan varmistaa, että sovellus antaa käyttäjälle virheilmoituksen.

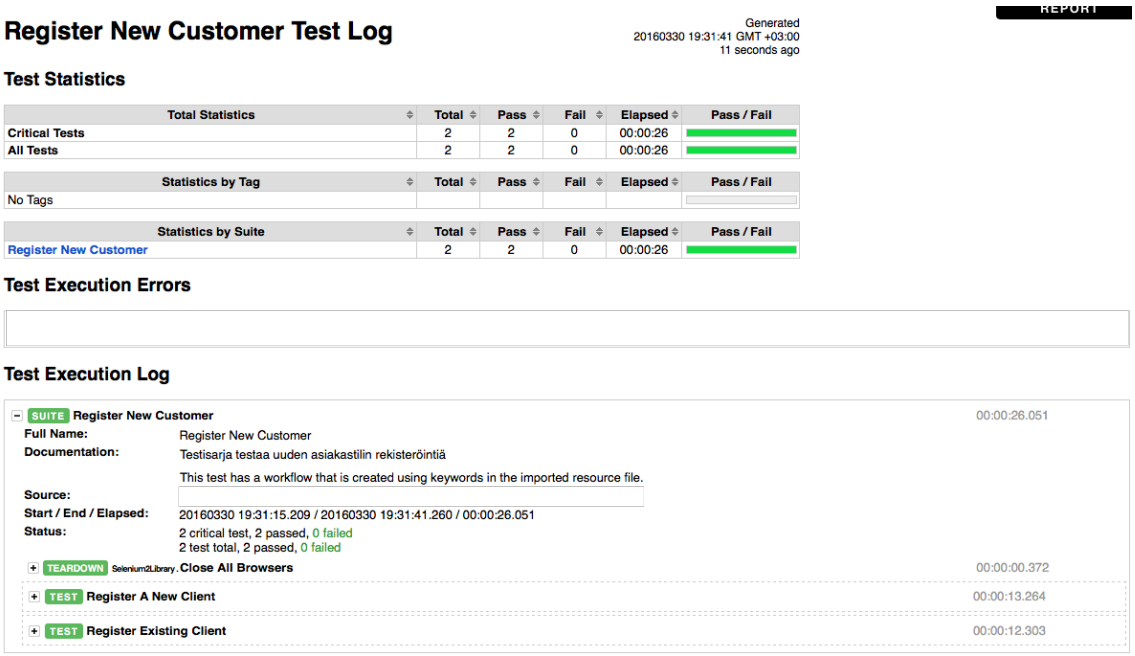
6.5 Onnistuneen testauksen analysointi

Tässä kohdassa analysoidaan kahta työssä onnistunutta testisarjaa, jotka ovat ”Uuden asiakastilin rekisteröinti” ja ”Kirjautuminen asiakas-sivuille”.

”Uuden asiakastilin rekisteröinti”-testisarja on onnistunut, kun kaikki testisarjan testitapaukset on saatu onnistuneesti läpi. Lopputuloksena sovellus navigoi rekisteröinnin jälkeen automaattisesti ”Oma tili”-sivulle ja näyttää loppukäyttäjälle, että tili on luotu.

”Kirjautuminen asiakastili-sivuille”-testisarja on onnistunut, kun kaikki testisarjan testitapaukset on saatu onnistuneesti läpi. Lopputuloksena sovellus navigoi kirjautumisen jälkeen ”Oma tili”-sivulle, jonka jälkeen kirjaudutaan sovelluksesta onnistuneesti ulos.

Kuvassa 20 näkyy Robot Frameworkin luoma lokitiedosto suoritetun testauksen jälkeen, mistä huomataan, että testisarja on suoritettu onnistuneesti. Testisarja sisältää kaksi testitapausta, jotka ovat ”Uuden asiakkaan rekisteröinti” ja ”Olemassa olevan asiakkaan rekisteröinti”.



Register New Customer Test Log

Generated
20160330 19:31:41 GMT +03:00
11 seconds ago

REPORT

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	2	2	0	00:00:26	
All Tests	2	2	0	00:00:26	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Register New Customer	2	2	0	00:00:26	

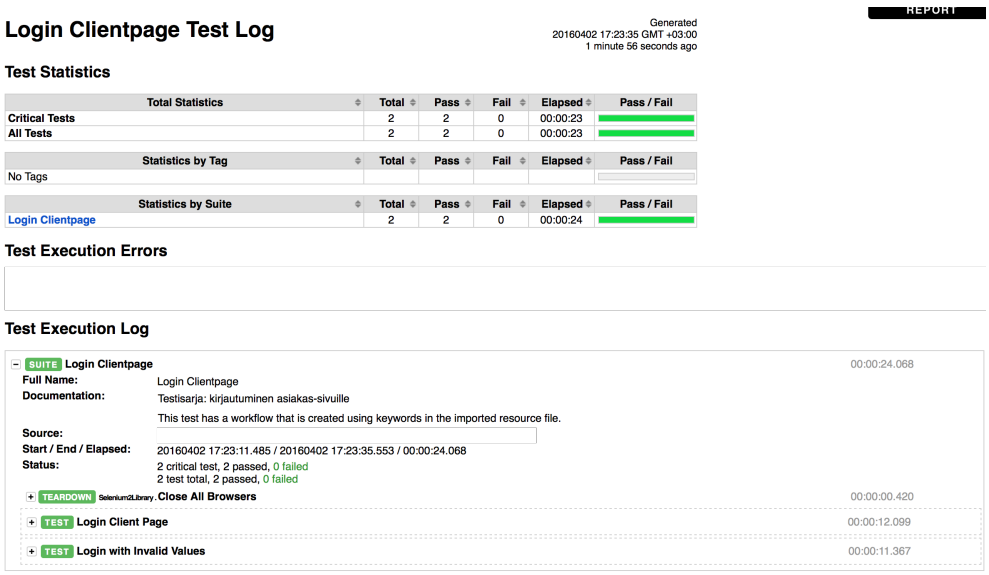
Test Execution Errors

Test Execution Log

SUITE Register New Customer 00:00:26.051
Full Name: Register New Customer
Documentation: Testisarja testaa uuden asiakastilin rekisteröintiä
This test has a workflow that is created using keywords in the imported resource file.
Source:
Start / End / Elapsed: 20160330 19:31:15.209 / 20160330 19:31:41.260 / 00:00:26.051
Status: 2 critical test, 2 passed, 0 failed
2 test total, 2 passed, 0 failed
TEARDOWN Selenium2Library . Close All Browsers 00:00:00.372
TEST Register A New Client 00:00:13.264
TEST Register Existing Client 00:00:12.303

Kuva 20. *Uuden asiakkaan rekisteröinti*

Lokitiedostosta voidaan myös huomata testisarjan suoritusaika. Koko testisarjan suoritus kesti noin 26 sekuntia, josta uuden asiakkaan luominen kesti vähän yli 13 sekuntia ja olemassa olevan noin 12 sekuntia. Testisarja testattiin 10 kertaa onnistuneesti ja huomattiin, että keskiarvo kokonaissuoritusajalle oli 26,5 sekuntia. Kun testaus suoritettiin manuaalisesti 10 kertaa, saatiin suoritusajan keskiarvoksi 1 minuutti ja 14 sekuntia.



Login Clientpage Test Log

Generated
20160402 17:23:35 GMT +03:00
1 minute 56 seconds ago

REPORT

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	2	2	0	00:00:23	
All Tests	2	2	0	00:00:23	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Login Clientpage	2	2	0	00:00:24	

Test Execution Errors

Test Execution Log

SUITE Login Clientpage 00:00:24.068
Full Name: Login Clientpage
Documentation: Testisarja: kirjautuminen asiakas-sivuille
This test has a workflow that is created using keywords in the imported resource file.
Source:
Start / End / Elapsed: 20160402 17:23:11.485 / 20160402 17:23:35.553 / 00:00:24.068
Status: 2 critical test, 2 passed, 0 failed
2 test total, 2 passed, 0 failed
TEARDOWN Selenium2Library . Close All Browsers 00:00:00.420
TEST Login Client Page 00:00:12.099
TEST Login with Invalid Values 00:00:11.367

Kuva 21. *Kirjautuminen asiakas-sivuille*

Kuvassa 21 on kuvankaappaus testisarjasta, josta testaan asiakkaan kirjautumista asiakas-sivulle. Siitä huomataan myös, että testisarja on suoritettu onnistuneesti ja se sisältää

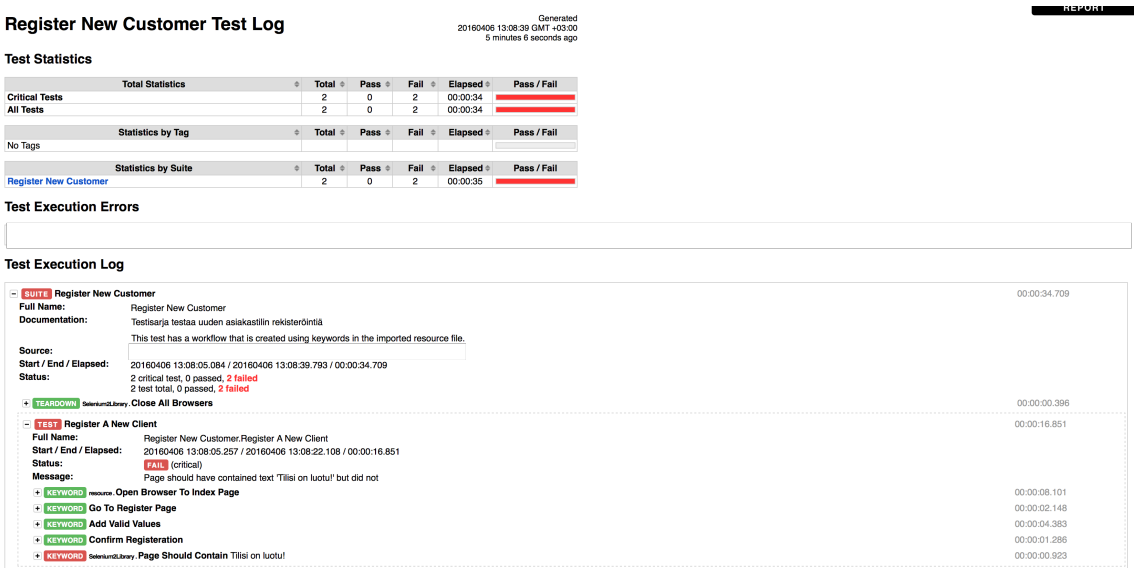
kaksi testitapausta, jotka ovat kirjautuminen asiakas-sivuille ja kirjautuminen asiakas-sivulle kelvottomilla tunnuksilla.

Testisarjan suoritusaika on noin 24 sekuntia, josta kirjautuminen asiakas-sivulle kesti noin 12 sekuntia ja kirjautuminen kelvottomilla tunnuksilla vähän yli 11 sekuntia. Tämä testisarja suoritettiin onnistuneesti 10 kertaa ja koko testisarjan suoritusajan keskiarvo oli 22,4 sekuntia. Testaus suoritettiin myös manuaalisesti 10 kertaa ja suoritusajan keskiarvoksi saatiin 38,2 sekuntia.

Manuaalinen testauksen on suorittanut testaaja, joka tuntee testattavan sovelluksen ja testitapaukset. Vaikka manuaalisessa testauksessa ei tapahtunut kirjoitusvirheitä ja virheellisiä klikkauksia, oli automaatiotestaus melkein kaksi kertaa nopeampaa kuin manuaalinen testaus ”kirjautuminen asiakas-sivulle” testitapauksessa ja yli kolme kertaa nopeampaa ”uuden asiakastilin rekisteröinti” testitapauksessa. Testausta olisi myös mahdollista nopeuttaa entisestään pienentämällä Robot Frameworkin ”delay:ta” eli viivettä, joka määrittelee testauksen nopeuden. Kuitenkin viiveen pienentäminen lisäsi testauksen epäonnistuneita tapauksia.

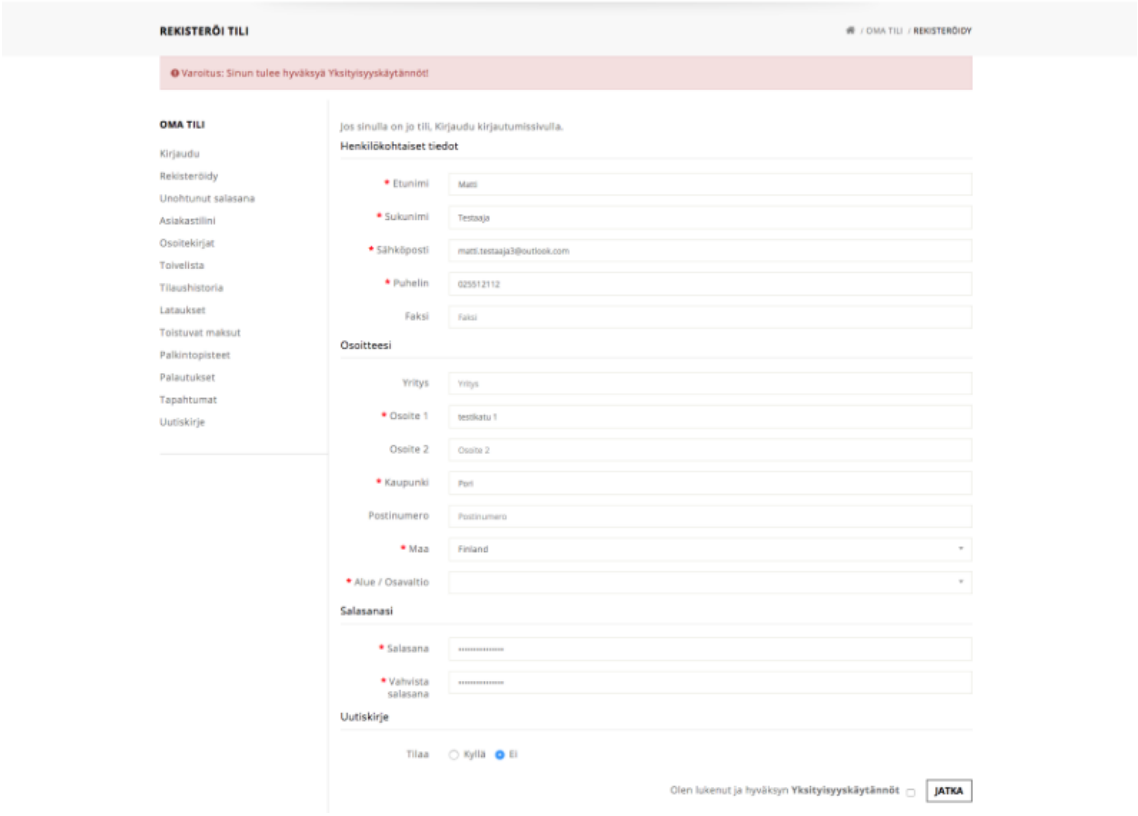
6.6 Epäonnistuneen testauksen analysointi

Testiautomaatio testejä ajattaessa kohdattiin myös ongelmia, mikä on hyvin yleistä ohjelmistoalalla. Ongelma ei ollut kuitenkaan testattavassa sovelluksessa vaan kirjoitetussa testiautomaatiossa. Tässä työssä suurimmat ongelmat syntyivät syötteiden ja valikoiden testauksessa sekä kuinka asettaa oikea viive testirobotille.



Kuva 22. *Uuden asiakkaan rekisteröinnin epäonnistunut testi*

Kuvassa 22 nähdään kuinka ”Uuden asiakkaan rekisteröinti” -testitapaus on epäonnistunut. Testi on mennyt hyvin melkein loppuun saakka, mutta lopussa rekisteröinnin varmistaminen on epäonnistunut. Ylläolevasta lokista voidaan huomata, että rekisteröinnin varmistuksessa olisi pitänyt tulla esille ”Tilisi on luotu!” ilmoitus käyttäjälle



Kuva 23. *Kuvankaappaus rekisteröinti-sivusta*

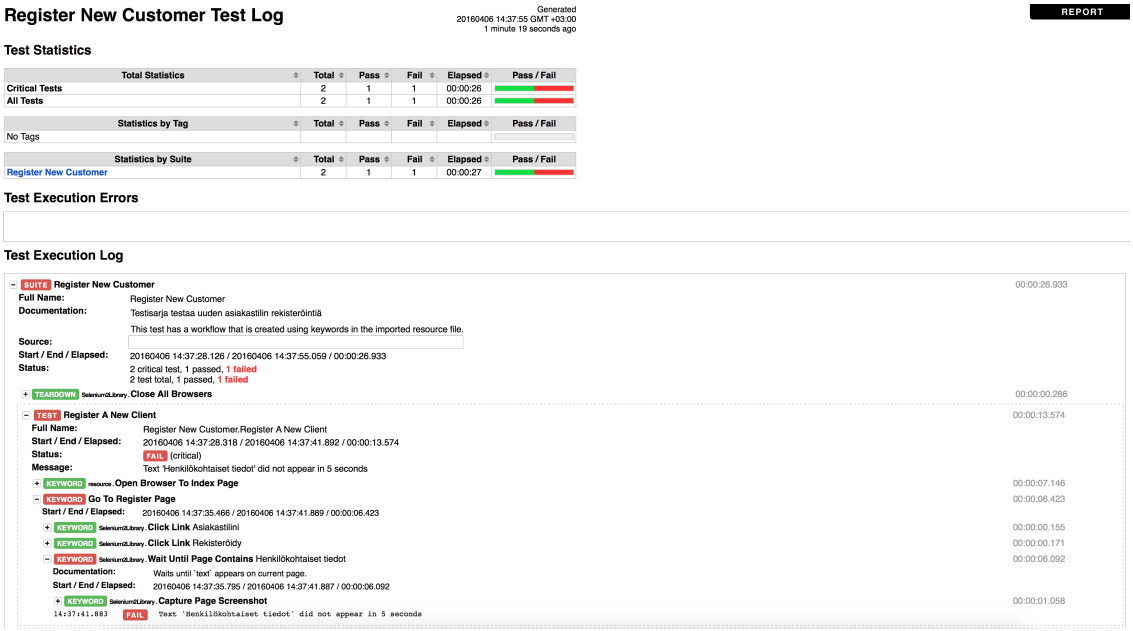
Epäonnistuneista testeistä Robot Framework luo kuvankaappauksen tapauksesta, jossa kohtaan testi on epäonnistunut. Kuvankaappauksen saa esille, kun avaa epäonnistuneen testistepin auki.

Kuvassa 23 on kuvankaappaus testattavasta sovelluksesta missä testi on epäonnistunut. Kuvasta huomataan, että rekisteröinti ei ole onnistunut, koska rekisteröinnin jälkeen sovelluksen olisi pitänyt navigoida automaattisesti ”Oma Tili”-sivulle, mutta testauksessa se on jäänyt rekisteröinti-sivulle. Kuvasta 23 nähdään, että sovellusta antaa käyttäjälle ilmoituksen ”Sinun tulee hyväksyä yksityisyyskäytännöt”, josta voidaan päätellä, ettei testirobotti ole niitä hyväksynyt. Tämän jälkeen testiautomaatioon kirjoitettiin alla oleva komento, jonka jälkeen testi suoritettiin onnistuneesti.

```
select checkbox  agree
```

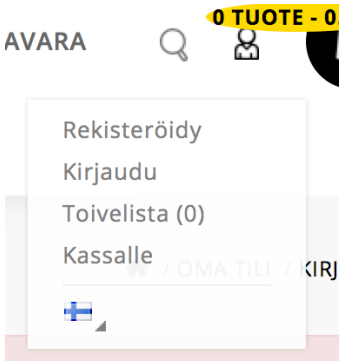
Robot Frameworkin viiveen eli ”delay:n” asettaminen oli toinen tapaus, joka aiheutti suuria ongelmia tässä työssä. Aluksi testiautomaatiotapauksia kirjoitettiin ja testattiin ilman

viivettä. Myöhemmässä vaiheessa huomattiin, että onnistuneet testit eivät enää menneet läpi joka kerta, mikä vaikeutti testiautomaation kirjoittamista ja testaamista.



Kuva 24. *Kuvankaappaus viive ongelman lokista*

Ongelma ei ollut helposti ratkaistavissa, koska lokeissa (Kuva 24) luki ainoastaan, että jokin elementti, sivun ilmoitus tai otsikko ei tullut näkyviin viiden sekunnin aikana. Aluksi ajateltiin sen johtuvan testiautomaation koodista, mutta myöhemmässä vaiheessa huomattiin testirobotin jättävän kirjoittamatta syötteitä. Tämän lisäksi testattava sovellus ei pysynyt testirobotin perässä perättäisissä linkeissä. Tällainen tapaus oli esimerkiksi navigointi etusivulta rekisteröinti-sivulle (Kuva 25).



Kuva 25. *Navigointi etusivulta rekisteröinti-sivulle*

Kun testirobotti klikkasi asiakastili-ikonia (Kuva 25) suurennuslasi-ikonin oikealta puolelta ja sen jälkeen rekisteröinti-linkkiä, niin testattava sovellus ei ehdi huomioimaan kahta perättäistä klikkausta. Tässä tapauksessa sen olisi pitänyt tauksen aikana siirtyä rekisteröinti-sivulle, mutta testi epäonnistui ja testirobotti loi kuvankaappauksen etusivusta.

Ongelma oli hankala ratkaista, koska se ei toistunut joka testauksen jälkeen, mutta lopulta Robot Frameworkin sivuilta löydettiin ongelmaan ratkaisu. Se saatiin ratkaistua asettamalla viive ”Set Selenium Speed”-komennon avulla. Viiveeksi asetettiin 0.1, joka riitti ajamaan testit onnistuneesti läpi ilman testien suoritusajan hidastumista.

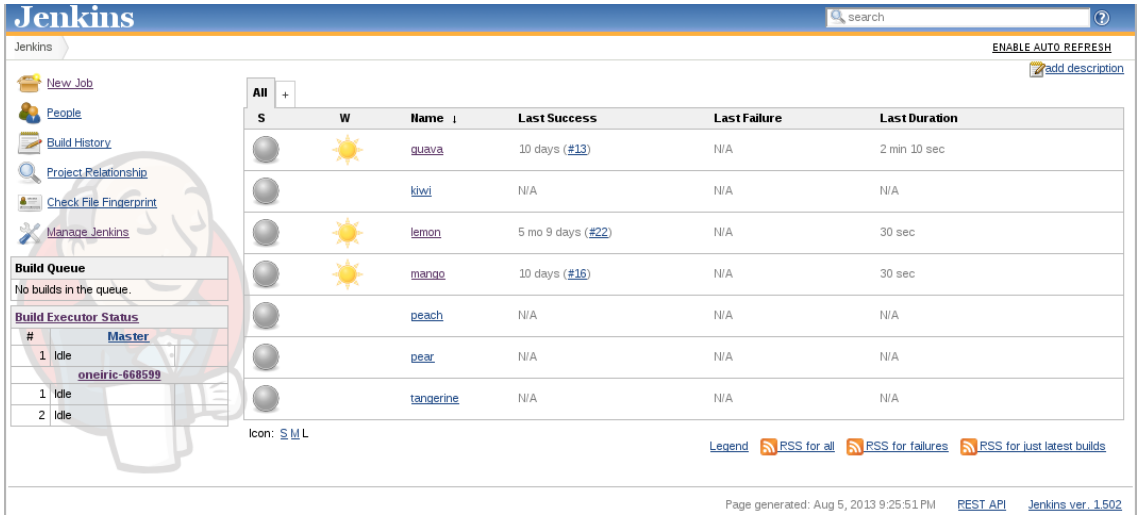
7. JATKOKEHITTELY

Tämän työn toteutusvaiheen aikana syntyi monia uusia ideoita ja ajatuksia, miten tässä työssä syntynyttä lopputulosta pystyttäisiin kehittämään vielä enemmän ja viemään uudelle tasolle. Valitettavasti kehitystyön aika oli rajallinen, joten niiden toteuttaminen jää tulevaisuuteen.

Ensimmäinen kehitysidea on vähentää testeissä syntyvää toistettavuutta ja yhdistää useasti käytettyjä muuttujia yhteen testitiedostoon, mistä niitä voidaan kutsua. Tämän lisäksi olisi hyvä lisätä avainsanojen käyttöä testeissä, jolloin niistä tulisi helpommin luettavia ja nopeuttasi testien kirjoittamista.

Suurin kehitysidea on kuitenkin viedä automaatiotestit Jenkins CI-serverille. Jenkins on johtavista avoimen lähdekoodin automaatiopalvelimista. Se tukee satoja eri liitännäisiä ja sen avulla voi esimerkiksi kääntää koodia ja ajaa automaatiotestejä automaattisesti (Jenkins, 2016).

Jenkinsiin luodaan erilaisia jobeja, joita se ajaa joko käyttäjän pyynnöstä tai automaattisesti. Kuvassa 25 on kuvankaappaus Jenkinsin etusivusta, jossa on useita eri jobeja luotuna. Jos job on onnistunut, muuttuu sen vieressä oleva pallo vihreäksi ja taas epäonnistuessaan punaiseksi.



Kuva 26. *Kuvankaappaus Jenkins CI-palvelimesta (Jenkins, 2016)*

Tarkoituksena on luoda tässä työssä syntyneistä automaatiotesteistä Jenkins jobit, joita se ajaa aina jokaisen uuden versiopäivityksen jälkeen. Näin ollen automaatiotestit pysyvät hyvin hallinnassa, ja niiden ajaminen on hyvin yksinkertaista ja nopeaa.

8. YHTEENVETO

Tämän diplomityön tavoitteena oli automatisoida web-sovelluksen regressiotestaus, jotta ohjelmiston laadun varmistaminen olisi nopeampaa ja laadukkaampaa kuin manuaalisesti testattuna. Tavoite onnistui ja automatisoidut testit ovat jo käytössä regressiotestauksessa.

Työn toisessa luvussa käsiteltiin testausta yleisellä tasolla ja esiteltiin testaukseen liittyviä käsitteitä. Luvussa kolme käsiteltiin regressiotestaus ja pohdittiin missä tilanteissa sitä tulisi käyttää. Lyhyesti sanottuna regressiotestaus tulee suorittaa jokaisen uuden versiopäivityksen jälkeen, jolloin voidaan varmistaa, että sovellus toimii oikein myös uusien ominaisuuksien jälkeen. Tästä syystä regressiotestaus on hyvä kohde automatisoinnille.

Neljännessä luvussa käsiteltiin testauksen automatisointia ja esiteltiin perusteluja, koska testaus olisi hyvä automatisoida. Tämän lisäksi luvussa esiteltiin eri työkaluja, joiden avulla testejä pystyy automatisoimaan. Robot Framework ja Selenium valittiin työkaluiksi, koska olivat ennestään tuttuja.

Luvussa viisi käsiteltiin tarkemmin Robot Frameworkia ja sen arkkitehtuuria. Tämän lisäksi luvussa käytiin läpi Robot Frameworkin, Seleniumin ja PyCharmin asennukset ja millaisia ongelmia syntyi asennuksien aikana. Suurimmat ongelmat syntyivät oikean Pythonin ja Robot Framework versioiden asennuksessa. Aluksi yritettiin asentaa Robot Frameworkin 2.9 versioita Pythonin 2.6 version avulla, mutta testejä ei pystynyt suorittamaan, koska Robot Frameworkin versio 2.9 tarvitsi vähintään Pythonista versiota 2.7. Tämän lisäksi ongelmia tuli Pythonin asennuksen kanssa, koska käyttöjärjestelmässä oli valmiiksi asennettuna versio 2.6. OS X-käyttöjärjestelmän paketinhallintajärjestelmän avulla Pythonin sai helposti päivitettyä versioon 2.7.

Luvun viisi loppupuolella käytiin läpi testien kirjoittamista ja ajamista sekä käytiin läpi yksityiskohtaisesti PyCharmin käyttöliittymää. PyCharmilla oli helppo kirjoittaa testejä, koska se tuki Robot Frameworkia ja näin ollen ilmoitti käyttäjälle virheistä. Tämän lisäksi luvussa käytiin läpi, mitä Robot Frameworkin lokitiedosto sisältää testin suorituksen jälkeen. Lokitiedostot ovat helppolukuisia ja epäonnistuneen testin jälkeen ongelmakohdan löytää helposti punaisesta väristä lokissa.

Luku kuusi käsitteli diplomityössä syntynyttä tuotosta eli web-sovelluksen regressiotestauksen automatisointia. Työ sisälsi kolme testitapausta, joista yhdestä esiteltiin käyttöja testitapaus. Esimerkiksi valittiin uuden asiakkaan rekisteröinti, minkä esitietoja tarvitaan muissa testitapauksissa. Tämän lisäksi luvussa analysoitiin onnistuneita ja epäonnistuneita testejä ja vertailtiin manuaali- ja automaatiotestausta. Eniten vaikeuksia tuottivat syötteiden ja valikoiden testaaminen sekä Robot Frameworkin viive testauksessa. Kun viiveen olemassaolo saatiin selville, niin testien ajaminen ja kirjoittaminen olivat helppoa.

Tästä työstä voidaan päätellä, että regressiotestauksen automatisointi on tarpeellista sekä ohjelmiston kehittäjien, että yrityksen kannalta. Automatisoitujen testien myötä ohjelmiston versiopäivitykset nopeutuvat ja laadunvarmistus paranee. On kuitenkin muistettava, että kaikkea ei pysty eikä tule automatisoida, vaan manuaalista testausta tarvitaan aina testiautomaation ohella.

LÄHTEET

Bach, J., (1999), Test Automation Snake Oil, Viitattu: 13.3.2016,
Saatavissa: http://www.satisfice.com/articles/test_automation_snake_oil.pdf

Binder, R., (1999), Testing object-oriented systems, Addison-Wesley, s. 761-768.

Dustin, (2002), Effective Software Testing, Viitattu 10.3.2016,
Saatavissa:
http://www.cse.hcmut.edu.vn/~hiep/KiemthuPhanmem/Tailieuthamkhao/Effective Software Testing - 50 specific ways to improve your testing.pdf

Framework, R., (2015), Intruction,
Viitattu: 8.2.2016, Saatavissa: http://robotframework.org

Gerrard, P., (1997), Testing GUI Applications. EuroSTAR'97, (November), Viitattu: 27.2.2016,
Saatavissa: http://gerrardconsulting.com/sites/default/files/Techgui.pdf

Haikala, I., & Märijärvi, J., (2006), Ohjelmistotuotanto, Helsinki : Talentum, s.287-290

Harrold, M. J., Gupta, R. & Soffa, M. L., (1993), Driving Software Quality: How Test-Driven Development Impacts Software Quality, s.270-285

Jenkins, (2016), Viitattu: 12.4.2016,
Saatavissa:https://wiki.jenkins-ci.org/download/attachments/66846870/snapshot7.png?version=1&modificationDate=1375741040000

Katara, M., (2011), Hyväksymistestaus, Viitattu: 14.1.2016,
Saatavissa:http://www.cs.tut.fi/~testaus/s2011/luennot/OHJ-3060_2011_110-170.pdf

Kit, E., (1995), Software Testing in the Real World Improving the Process, Addison-Wesley, s. 57-61

Klärck, P., (2009), Experience, Viitattu: 27.2.2016,
Saatavissa: http://eliga.fi/index.html

Korjus, N., (2015), Ohjelmistotestauksen perusteet, Viitattu: 23.1.2016,
Saatavissa:
https://noppa.lut.fi/noppa/opintojakso/ct60a4150/materiaali/ohjelmistotestauksen_perusteet_-manuaali.pdf&usg=AFQjCNHr-oLsP

Koskela, L., (2007), Test Driven: Practical TDD and Acceptance TDD for Java Developers, Manning Publications Co, s. 44

Leung, H. K. N., White, L., (1991), A Cost Model to Compare Regression Test Strategies. Los Alamitos: IEEE Computer Society, s. 201-20

Luukkainen, M., (2009), Ohjelmistojen mallintaminen, Viitattu: 16.1.2016, Saatavissa: https://www.cs.helsinki.fi/u/mluukkai/ohmas10/luentokalvot/luento2.pdf

Marick, B., (2000), When Should a Test Be Automated?, Viitattu: 23.1.2016 Saatavissa: http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=ART&ObjectId=2010#authorbio

Mittal, N. & Acharya, I., (2003), An Open Framework for Managed Regression Testing. International Conference, Testcom, Sophia Antipolis, France. s. 265–278.

Myers, G. J., (2012), The Art Of Software Testing, Viitattu: 14.1.2016, Saatavissa: http://www.computing.dcu.ie/~ray/teaching/CA358/TheArtofSoftwareTesting.pdf

Nokia Solution and Networks, (2015), Robot Framework User Guide, Saatavissa: http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html

Pan, J., (1999), Software Testing, Viitattu: 16.1.2016, Saatavissa: http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/

Patton, R., (2001), Software Testing, Sams, s.54-57

Rothermel, G. & Harrold, M. J. (1996), Analyzing Regression Test Selection Techniques, IEEE Transactions on Software Engineering, s. 529-551

Sahipro, (2016), Sahi Open Source Automation Testing Tool For Web Applications, Saatavissa: www.sahipro.com

Santos, T., (2009), Installation instructions, Viitattu 23.3.2016, Saatavissa: https://github.com/robotframework/robotframework/blob/master/INSTALL.rst

Selenium HQ, (2016), What is Selenium?, Viitattu 29.3.2016 Saatavissa: http://www.seleniumhq.org

Smartbear, (2016), Automated Software Testing, Viitattu: 22.2.2016, Saatavissa: https://smartbear.com

Taft, D., (2010), JetBrains Strikes Python Developers with PyCharm 10, Saatavissa:http://www.eweek.com/c/a/Application-Development/JetBrains-Strikes-Python-Developers-with-PyCharm-10-IDE-304127

Tamres, L., (2002), Introducing Software Testing, Harlow: Pearson Education Ltd, s. 223-226

Whittaker, J. A., (2000), What Is Software Testing? And Why Is It So Hard?, Viitattu: 15.1.2016,
Saatavissa:
http://faculty.washington.edu/ajko/teaching/info461/media/Whittaker2000SoftwareTesting.pdf

Wong, W. E., Horgan, J. R., London, S., Agrawal, H., & Street, S., (1997), A Study of Effective Regression Testing in Practice, s. 264–274.

Zambelich, K., (1998), Totally data-driven automated testing, Viitattu: 14.2.2016
Saatavissa:
http://www.oio.de/public/softwaretest/Totally-Data-Driven-Automated-Testing.pdf